Chapitre 6

La création des objets en JAVA

L'objectif de ce chapitre est de montrer comment sont créés, manipulés et organisés en mémoire, les objets du langage JAVA et des tableaux.

1. Le cycle de vie d'un objet 2
1.1. La création d'un objet : new 2
1.2. Un objet JAVA est un pointeur 2
1.3. La destruction d'un objet 5
2. L'accessibilité aux objets 5
2.1. Le passage en paramètre d'un objet 5
2.2. La visibilité des attributs 6
3. Les constructeurs d'une classe 10
3.1. Définition 10
3.2. Le constructeur par défaut d'une classe 12
3.3. Le remplacement du constructeur par défaut 12
3.4. Les constructeurs avec des paramètres 15
3.5. L'erreur commune sur le constructeur 18
4. Le type tableau en Java 20
4.1. Introduction 20
4.2. Définition 20
4.2.1. La déclaration 20
4.2.2. L'allocation 20
4.2.3. La gestion de la mémoire du programme 21
4.2.4. L'utilisation 23
4.2.5. Débordement d'un tableau 24
4.2.6. Raccourci 24
4.2.7. La boucle for énumérative 25
4.3. Les tableaux de String 25
4.4. L'initialisation des tableaux 26
4.5. Les tableaux à 2 dimensions 26
4.6. Les tableaux à N dimensions 28
5. La classe Arrays 28

1. Le cycle de vie d'un objet

1.1. La création d'un objet : new

La création d'un objet se fait en instanciant une classe.

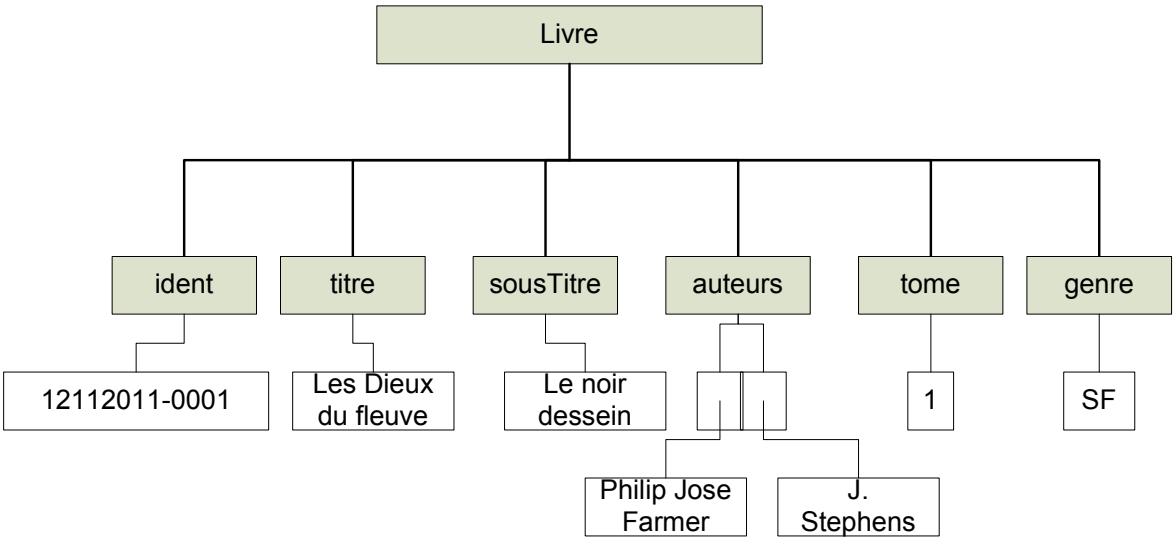
Un objet est une instance d'une classe.

L'instruction **new** permet de créer un objet à partir d'une classe.

L'objet qui résulte du processus d'instanciation contient toute l'arborescence de composition des attributs définis dans la classe.

Une fois instancié, l'accès aux attributs se fait par l'opérateur : . (point).

Si on prend l'exemple 5 du chapitre précédent, on obtient, l'arborescence suivante :

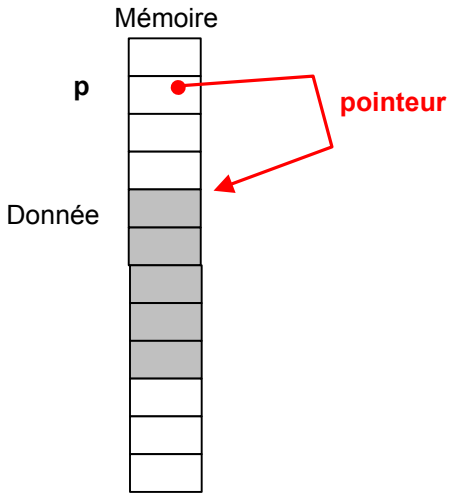


L'allocation en mémoire d'un objet correspond exactement à l'arborescence des données de son arbre de composition.

1.2. Un objet JAVA est un pointeur

Un objet en JAVA est un **pointeur**.
La notion de pointeur est une notion qui existe depuis tous les premiers langages informatiques et très répandue.

Définition : Un pointeur est une adresse mémoire à laquelle se trouve une donnée informatique. Cela peut se présenter en mémoire de l'ordinateur de la manière suivante :



p est ici une variable informatique qui est un pointeur. On dit que p pointe sur la Donnéee.

C'est-à-dire que p contient l'adresse mémoire de Donnée.

Ainsi toute variable ou attribut de type de classe (type "référence) est un pointeur. Tout objet est un pointeur et Donnée contient tous les attributs de l'objet.

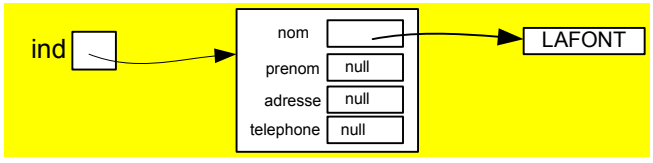
En Java cela revient à faire :

```
<classe> p = new <classe>(    )
```

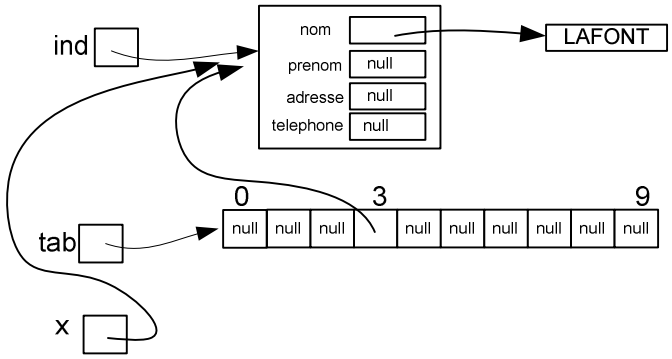
```
public class Pointeur
{
    public static void main(String args[])
    {

        Individu ind = new Individu();
        ind.nom = "LAFONT";


        Individu[] tab = new Individu[10];

        tab[3] = ind;

        Individu  x = tab[3];

        x.nom = "DURAND";

        Terminal.ecrireStringln(ind.nom);     // DURAND !!!
        Terminal.ecrireStringln(tab[3].nom);  // DURAND !!!
    }
}

public class Individu
{
     int     age ;
     String nom;
     String prenom;
     Adresse adresse;
     String telephone;
}
```

Détails et comportement de la mémoire :

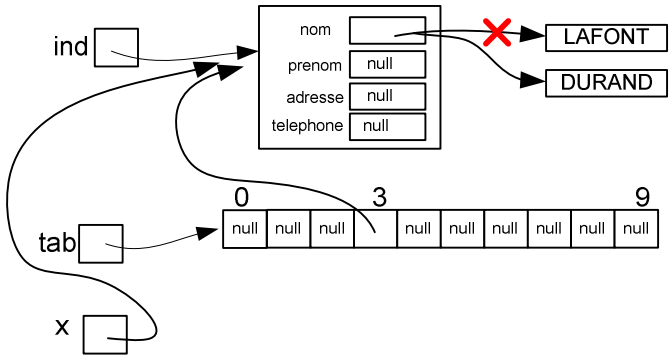
```
        Individu ind = new Individu();
        ind.nom = "LAFONT";   // équivalent à   new String()
```



```
Individu[] tab = new Individu[10];

tab[3] = ind;

Individu  x = tab[3];
```




```
x.nom = "DURAND";
```



1.3. La destruction d'un objet

Afin de ne pas encombrer inutilement la mémoire du programme objet, il est indispensable de détruire les objets alloués dont on a plus besoin.

Deux aspects ont pendant longtemps étaient mal maitrisés dans les langages orientés objet et qui ont été source d'erreur d'exécution :

- quand doit-on détruire les objets ?
- comment peut-on être sur que l'objet que l'on détruit n'est pas encore utilisé ?

JAVA répond à ces deux questions en prenant en charge automatiquement la destruction des objets qui ne sont plus utilisés.
Ce mécanisme s'appelle : le garbage-collector (ou ramasse miettes) ou gc.

2. L'accessibilité aux objets

2.1. Le passage en paramètre d'un objet

Le langage JAVA ne faisant que du passage par valeur de ses paramètres, fait passer un objet par valeur. Mais comme, un objet est un pointeur, **c'est le pointeur qui**

est passé en paramètre. Il est donc possible de modifier le contenu de l'objet passé en paramètre.

Exemple :

```
public class ExempleParamObjet
{
    public static void main(String args[])
    {
        Individu ind1 = new Individu();
        ind1.nom = "DUPONT";
        ind1.prenom  = "Michel";

        initAdresse(ind1,"22 rue du Pont");

        Terminal.ecrireStringln(ind1.adresse);  // 22 rue du Pont
    }

    static void initAdresse(Individu ind,String addr)
    {
        ind.adresse = addr;
    }
}

public class Individu
{
     int     age ;
     String nom;
     String prenom;
     Adresse adresse;
     String telephone;
}
```

Nous verrons que cette façon d'initialiser les attributs d'un objet est un usage peu courant.

2.2. La visibilité des attributs

Les attributs d'une classe peuvent être de différents genres:
- attributs privées
- attributs publics
- attributs protégés
- attributs défauts

Le rôle du genre de l'attribut est de protéger l'accès de l'attribut d'un classe par du code exécuté dans une autre classe.

Pourquoi protéger les attributs d'une classe ?

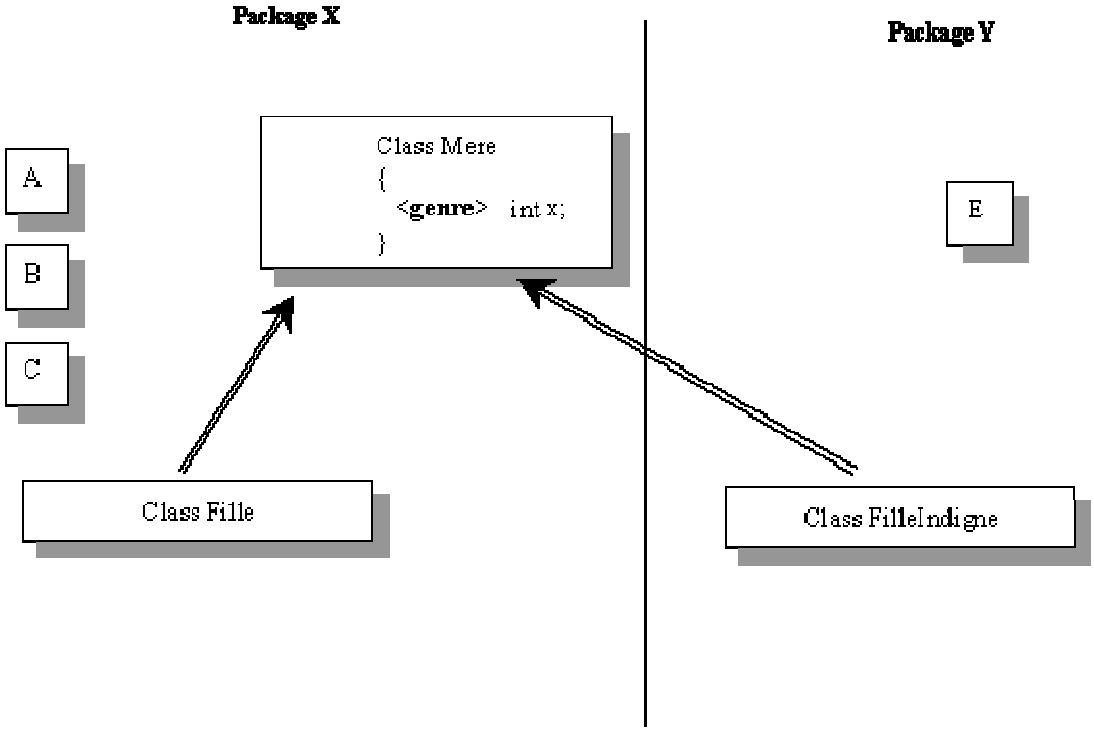
Les attributs d'une classe représentent la structure d'un objet, les valeurs de ses attributs forment la cohésion de l'objet. La classe doit donc garantir la cohésion de ces valeurs en protégeant leurs affectations par le code extérieur à la classe.

Par exemple, les attributs privés ne peuvent être modifiés que par les méthodes de la classe alors que les attributs publics peuvent être modifiés directement par les méthodes des autres classes.

De plus, rendre tous les attributs d'une classe privés permet de cacher l'implémentation de la classe et permet ainsi de faire évoluer la structure interne de la classe en gardant la compatibilité des méthodes utilisés par l'extérieur.

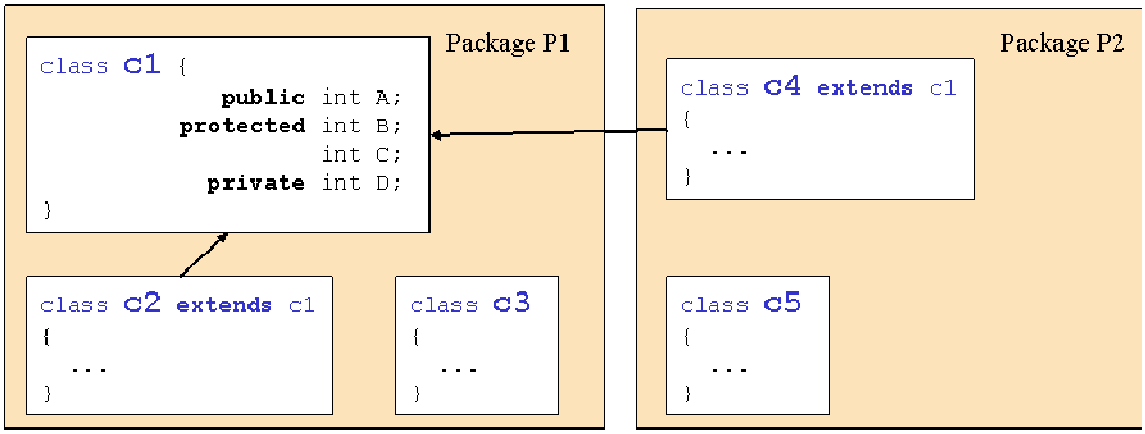
Package = Répertoire

ClasseFille et ClasseFilleIndigne hérite de Mere



genre	x est visible dans les classes
private	Mere
public	Toutes
protected	Mere, Fille, FilleIndigne, A, B, C
default *(vide)*	Mere, Fille, A, B, C

Une autre représentation :



	A	B	C	D
Accessible par c1	oui	oui	oui	oui
Accessible par c2	oui	oui	oui	-
Accessible par c3	oui	oui	oui	-
Accessible par c4	oui	oui	-	-
Accessible par c5	oui	-	-	-

Exemple : Répertoire **Scope** des exemples contenant les fichiers : Exemple37 sur le site

Scope/
- **Scope.java**
- **pkgX/**
  - **A.java**
  - **B.java**
  - **C.java**
  - **Mere.java**
  - **Fille.java**
- **pkgY/**
  - **FilleIndigne.java**
  - **E.java**

Fichier: Mere.java

```
public class Mere
{
    public    int x_public;
    private   int x_private;
    protected int x_protected;
    int           x_default;

    public Mere()
    {
       x_private = 100;
    }

    public int getXPrivate() { return x_private; }
}
```

Fichier: A.java

```
package pkgX;

public class A
{
```

```
      public A()
      {
          Mere m = new Mere();
          int n;
          n = m.x_public;
          //n = m.x_private;  // x_private has private access in pkgX.Mere
          n = m.x_protected;
          n = m.x_default;
      }

  }
```

Fichier: Fille.java

```
  package pkgX;

  public class Fille extends Mere
  {
      public Fille()
      {
          Mere m = new Mere();
          int n;
          n = m.x_public;
          //n = m.x_private;  // x_private has private access in pkgX.Mere
          //n = x_private;    // x_private has private access in pkgX.Mere
          n = m.x_protected;
          n = x_protected;
          n = m.x_default;
          n = x_default;
      }
  }
```

Fichier: FilleIndigne.java

```
  package pkgY;

  import pkgX.*;

  public class FilleIndigne extends Mere
  {
      public FilleIndigne()
      {
          Mere m = new Mere();
          int n;
          n = m.x_public;
          //n  =  m.x_private;       //  x_private  has  private  access  in
pkgX.Mere
          //n  =  x_private;          //  x_private  has  private  access  in
pkgX.Mere
          //n  =  m.x_protected;   //  x_protected  has  protected  access  in
pkgX.Mere
          n = x_protected;
          //n  =  m.x_default;   //  x_default  is  not  public  in  pkgX.Mere;
cannot be accessed from outside package
          //n = x_default;  // x_default is not public in pkgX.Mere; cannot
be accessed from outside package
      }

  }
```

Fichier: E.java

```
  package pkgY;
```

```
  import pkgX.*;

  public class E
  {
      public E()
      {
          Mere m = new Mere();
          int n;
          n = m.x_public;
          //n  =  m.x_private;       //  x_private  has  private  access  in
pkgX.Mere
          //n  =  m.x_protected;   //  x_protected has  protected  access  in
pkgX.Mere
          //n  =  m.x_default;   //  x_default  is  not  public  in  pkgX.Mere;
cannot be accessed from outside package
      }
  }
```

Nous verrons dans le paragraphe sur l'héritage qu'une classe qui hérite d'une autre classe hérite des attributs et des méthodes de la classe héritée.

Cette définition est la base de la définition de l'héritage.

3. Les constructeurs d'une classe

Les exemples qui suivent sont issus de :

Voir sur le site http://jacques.laforgue.free.fr l'exemple **Exemple09_Biblio**

3.1. Définition

Le rôle de tout constructeur est de créer un objet d'une classe et d'initialiser les attributs de l'objet.

Le constructeur est une méthode particulière de la classe. Cette méthode est **public** (Elle est privée dans des cas très particulier comme le cas du "singleton").

Elle a toujours pour nom, le **nom de la classe**.

La méthode **n'indique pas de type de valeur de retour** car c'est toujours le type de la classe.

On peut définir plusieurs constructeurs différents, du moment qu'ils se différencient par les paramètres de la méthode.

C'est l'instruction **new** qui appelle un des constructeurs de la classe.

Syntaxe de déclaration d'un constructeur :

```
public <nom de la classe> ( <paramètres> )
{
     // Code du constructeur
}
```

Le constructeur est donc une méthode comme une autre, elle exploite tous les attributs de l'objet, elle peut déclarer des variables locales, peut appeler des méthodes privées de la classe et des méthodes d'autres classes.

3.2. Le constructeur par défaut d'une classe

Le constructeur par défaut d'une classe est le constructeur utilisé par l'instruction new quand le programmeur n'a créé aucun constructeur pour la classe.

Ce constructeur alloue les attributs de l'objet et initialise par défaut les attributs avec les valeurs par défaut de Java.

Exemple :

```
public class Livre
{
    public String     ident;
    public String     titre;
    public String     sousTitre;
    public String[]   auteurs;
    public int        tome;
    public String     lien;
    public GenreLivre genre;
}

enum GenreLivre {
        HISTOIRE,
        GEOGRAPIE,
        ROMAN,
        SF
}

public class ConstructeurDefaut
{
      public static void main(String args[])
      {
          Livre x = new Livre();
          // x.ident    ➔ null
          // x.titre    ➔ null
          // x.tome     ➔ 0
          // x.auteurs  ➔ null
          // x.lien     ➔ null
          // x.genre    ➔ null

          x.titre = « Pas de titre » ;  // L'attribut titre est public
      }
}
```

3.3. Le remplacement du constructeur par défaut

L'objectif de créer son propre constructeur est d'initialiser les valeurs des attributs avec des valeurs différentes des valeurs par défaut de Java.
Ce constructeur n'a pas de paramètre.

Exemple :

On ajoute dans la classe Livre, le constructeur :

```
public class Livre
{
    public String     ident;
```

```
    public String     titre;
    public String     sousTitre;
    public String[]   auteurs;
    int               tome; // 0 si pas plusieurs tome
    String            lien;
    GenreLivre        genre;

    public Livre()
    {
        ident = "";
        titre = new String("<Sans Titre>");
        sousTitre = "";
        auteurs = null;
        tome = 0;
        lien = null;
        genre = null;
    }

    static public void afficherLivres(Livre... livres)
    {
        for(Livre l:livres)
            Terminal.ecrireStringln( l.toString() );
    }

    public String toString()
    {
        String str="";
        str=str+"Ident       : "+ident+"\n";
        str=str+"Titre       : "+titre+"\n";
        str=str+"Sous-titre  : "+sousTitre+"\n";
        str=str+"Tome        : "+tome+"\n";
        str=str+"Auteurs     : ";
        if (auteurs!=null)
            for(String s:auteurs) str=str+s+"   ";
        str=str+"\n";
        str=str+"Lien        : "+lien+"\n";
        str=str+"genre       : "+genre+"\n";
        return ( str );
    }

}
```

```
public class Exemple6
{
      public static void main(String args[])
      {
          Livre x = new Livre();

          Livre.afficherLivres(x);
      }
}
```

Exécution :

```
Gestion d'une bilbliotheque
Ident       :
Titre       : <Sans Titre>
Sous-titre  :
Tome        : 0
Auteurs     :
```

```
  Lien        : null
  genre       : null
```

3.4. Les constructeurs avec des paramètres

L'objectif des constructeurs avec des paramètres est d'initialiser les attributs de l'objet avec des valeurs spécifiques passées en paramètre.

Exemple :

```
public class Livre
{
    public String     ident;
    public String     titre;
    public String     sousTitre;
    public String[]   auteurs;
    int               tome; // 0 si pas plusieurs tome
    String            lien;
    GenreLivre        genre;

    public Livre()
    {
        ident = "";
        titre = new String("<Sans Titre>");
        sousTitre = "";
        auteurs = null;
        tome = 0;
        lien = null;
        genre = null;
    }

    public Livre(String titre,
                 String sousTitre,
                 String[] auteurs,
                 int tome,
                 GenreLivre genre)
    {
        this.ident = "";
        this.titre = titre;
        this.sousTitre = sousTitre;
        this.auteurs = auteurs;
        this.tome = tome;
        this.lien = "";
        this.genre = genre;
    }

    public void afficherTitre()
    {
        if (! sousTitre.equals(""))
            {
                Terminal.ecrireStringln(titre + "(" + sousTitre + ")" );
                return;
            }
        Terminal.ecrireStringln(titre);
    }

    public int getTome()
    {
        return tome;
    }

    public void setTome(int a_tome)
    {
        tome = a_tome;
```

```
    }

    public void setTitre(String a_titre)
    {
        titre = a_titre;
       sousTitre = "";
    }

    public void setTitre(String a_titre,String a_sousTitre)
    {
        titre = a_titre;
        sousTitre = a_sousTitre;
    }

    public void setAuteurs(String[] a_auteurs)
    {
        auteurs = a_auteurs;
    }

    public String toString()
    {
        String str="";
        str=str+"Ident       : "+ident+"\n";
        str=str+"Titre       : "+titre+"\n";
        str=str+"Sous-titre  : "+sousTitre+"\n";
        str=str+"Tome        : "+tome+"\n";
        str=str+"Auteurs     : ";
        if (auteurs!=null)
            for(String s:auteurs) str=str+s+"   ";
        str=str+"\n";
        str=str+"Lien        : "+lien+"\n";
        str=str+"genre       : "+genre+"\n";
        return ( str );
    }

    public String[] getAuteurs()
    {
        return auteurs;
    }

    public void lier(Livre l)
    {
        lien = l.ident;
        l.lien = ident;
    }

    static public void lier(Livre l1, Livre l2)
    {
        l1.lien = l2.ident;
        l2.lien = l1.ident;
    }

    static public void afficherLivres(Livre... livres)
    {
        for(Livre l:livres)
            Terminal.ecrireStringln( l.toString() );
    }
}

enum GenreLivre {
        HISTOIRE,
```

```
        GEOGRAPIE,
            ROMAN,
            SF
}
```

```
public class Exemple6
{
    public static void main(String... a_args)
    {
        Terminal.ecrireStringln("Gestion d'une bilbliotheque");

        Livre l1 = new Livre();

        String[] l_auteurs = new String[2];
        l_auteurs[0] = "Philip Jose Farmer";
        l_auteurs[1] = "J. Stephens";

        Livre l2 = new Livre("Les Dieux du fleuve",
                             "Le noir dessein",
                             l_auteurs,
                             1,
                             GenreLivre.SF);

        Livre.afficherLivres(l1,l2);

    }
}
```

S'il existe au moins un constructeur avec des paramètres et pas de constructeur sans paramètres alors le constructeur par défaut n'est plus accessible

```
Si on met en commentaire le constructeur sans parametre

public class Livre
{
    public String     ident;
    public String     titre;
    public String     sousTitre;
    public String[]   auteurs;
    int               tome; // 0 si pas plusieurs tome
    String            lien;
    GenreLivre        genre;

    /*
    public Livre()
    {
        ident = "";
        titre = new String("<Sans Titre>");
        sousTitre = "";
        auteurs = null;
        tome = 0;
        lien = null;
        genre = null;
```

```
    }
    */

>>>>>
```

On obtient alors l'erreur de compilation suivante :

```
javac Exemple6.java
Exemple6.java:7: cannot find symbol
symbol  : constructor Livre()
location: class Livre
        Livre l1 = new Livre();
                   ^
1 error
```

Pourquoi ?

Parce que c'est un choix de conception que fait le programmeur de définir son ou ses propres constructeurs dont le rôle est de donner de valeurs bien initialisées aux attributs de l'objet, et non de permettre d'appeler le constructeur par défaut qui va donner des valeurs "nulles" aux attributs de l'objet. Cela serait donc un risque que l'on créerait par erreur un objet mal initialisé.

3.5. L'erreur commune sur le constructeur

Il existe une erreur assez communes non détectées par le compilateur et dont la détection visuelle n'est pas évidente.

L'erreur est de mettre **void** comme type de retour au constructeur sans parametre alors qu'il n'existe pas d'autres constructeurs définis.
Cela est assez commun car toute méthode a un type de retour, souvent void, et on peut mettre void sans le faire exprès.

Le résultat est que votre constructeur sans paramètres n'est pas reconnu par java comme un constructeur et il appelle donc son constructeur par défaut.
Du coup, vos attributs ne sont pas initialisés comme vous le vouliez et le programme peut faire une erreur d'exécution.

Exemple :

```
public class Erreur1Constructeur
{
    public static void main(String args[])
    {
        Exemple1 ex = new Exemple1();
        ex.tab[0] = 22;    //← Erreur d'exécution
    }
}

class Exemple1
{
    public int[] tab;

    public void Exemple1()
    {
        tab = new int[10];
    }
}
```

Exécution :

```
java Erreur1Constructeur
Exception in thread "main" java.lang.NullPointerException
 at Erreur1Constructeur.main(Erreur1Constructeur.java:7)
```

Explication :

La méthode *public **void** Exemple1()* n'est pas un constructeur mais une simple méthode de type void dont le nom est *Exemple1*. Il n'y a donc pas de constructeur défini. C'est donc le constructeur par défaut qui est utilisé dans l'instruction :

Exemple1 ex = new Exemple1(); Or le constructeur par défaut initiliase l'attribut tab à null.

L'instruction *ex.tab[0] = 22;* déclenche donc une erreur car le tableau n'a pas été alloué.

4. Le type tableau en Java

4.1. Introduction

Le type tableau est un type de donnée essentiel dans la programmation.
Mais remplacer par le ArrayList si besoin.
Il existe autant de type de tableau qu'il existe de type primitif d'élément de tableau.

En java, on gère les types de tableau suivant :
- tableau d'entier
- tableau de double ou de float
- tableau de char (à ne pas confondre avec la chaîne de caractère)
- tableau de booléen
- tableau d' Objet (et donc tableau de String)

Il existe 3 étapes dans la prise en compte du tableau, en Java :
- la **déclaration** du tableau
- l' **allocation** du tableau
- l' **utilisation** du tableau

4.2. Définition

4.2.1. La déclaration

Un tableau se déclare de deux manières possibles

```
<type élément>[]  <variable> ;
ou
<type élément>  <variable>[];
```

<type élément> :
est le nom du type des éléments du tableau (types primitifs, type références)

<variable> :
Nom de la variable

Exemples :

```
int[]  tab_int;       // un tableau d'entier:

double[] tab3;        // un tableau de double

String[]  tabstr:     // un tableau de String


=➔ tab_int est égal à null
```

4.2.2. L'allocation

Après avoir déclaré le tableau, il faut maintenant allouer (créer) le tableau en taille physique. Pour cela on définit le nombre max d'élément du tableau (ou longueur du tableau) suivant la syntaxe suivante :

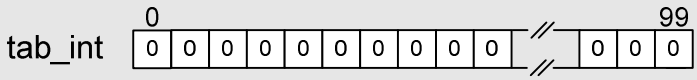
```
<variable tableau> = new <type élément>[ <taille> ]:
```

En Java, la taille d'un tableau peut être une variable et non uniquement une constante comme cela est le cas dans de nombreux langages.

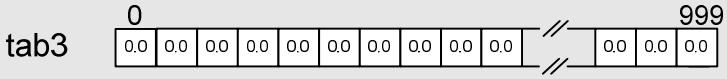
```
int n = Terminal.lireInt();
int[] tab = new int[n];
```

Exemples :

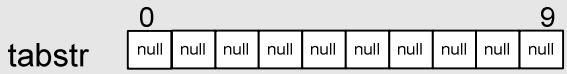
```
int[] tab_int;
tab_int = new int[100];
```




```
double[] tab3 = new double[1000];
```




```
String[] tabstr = new String[10];
```



Le tableau est alors initialisé avec des valeurs par défaut. La valeur par défaut utilisée est celle du type de l'élément :

int	:	0
double	:	0.0
float	:	0.0
char	:	'\0'
boolean	:	false
String	:	**null**

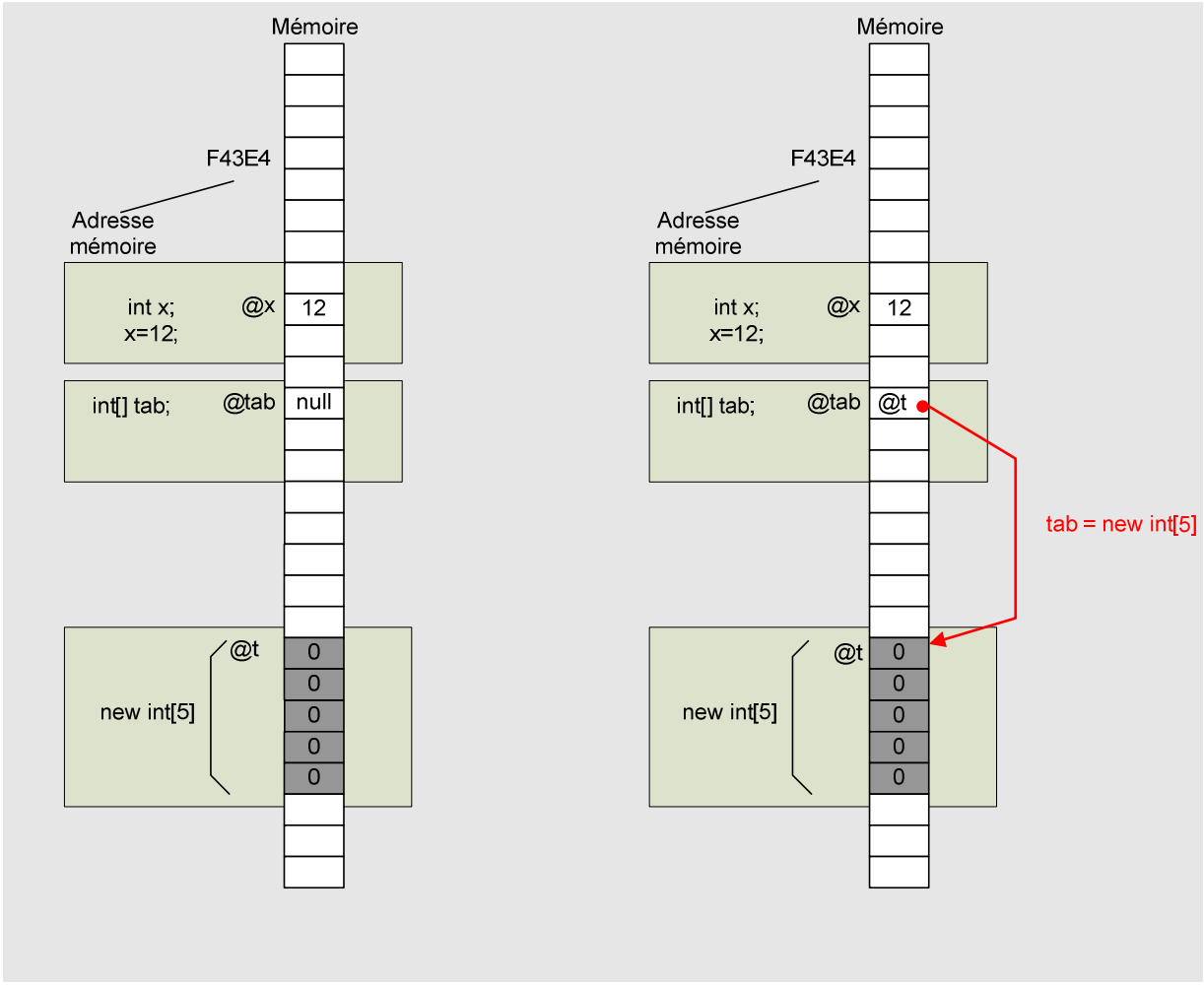
Les valeurs par défaut d'un tableau de String est que chaque case ne contient pas une chaîne mais la valeur *null* et non pas une chaine vide.

L'instruction **new** :
Cette instruction est utilisée dans le langage Java pour créer un tableau ou un objet.

4.2.3. La gestion de la mémoire du programme

Il est tant d'expliquer comment la mémoire du programme se comporte afin de comprendre comment les tableaux sont gérés.

Chaque variable déclaré dans un langage informatique (en Java la mémoire du programme est un espace mémoire géré par la JVM) prend une certaine place dans la mémoire :



Commentaires :

```
int x;
```

Quand on déclare la variable x, le programme alloue une zone mémoire devant contenir un entier. La variable x correspond donc à une adresse dans la mémoire @x.

Cela, pour une simple raison, qu'il est plus facile de manipuler ces zones d'adresse mémoire grâce à des identifiants. Il y a une correspondance entre l'adresse logique (la variable) et l'adresse physique (l'adresse mémoire).

```
x=12;
```

Cette instruction correspond à mettre la valeur 12 dans la zone mémoire qui se trouve à l'adresse @x

```
int[] tab;
```

Cette instruction correspond, comme précédemment, à la déclaration d'une variable. Comme pour la déclaration de l'entier, cela correspond à allouer une zone mémoire mais devant contenir l'adresse d'une autre zone mémoire (on appelle cela un **pointeur**).

Par défaut sa valeur est **null**.

```
new int[5]
```

Cette instruction **alloue** dynamiquement une zone mémoire devant contenir 5 entiers contigus dans la mémoire.

Cette zone mémoire se trouve à une adresse mémoire @t.

```
tab = new int[5];
```

Cette instruction consiste à affecter à la variable tab, l'adresse @t. Ainsi, tab contient l'adresse d'une autre donnée. On appelle cela un pointeur.

En java, les tableaux sont des pointeurs.

4.2.4. L'utilisation

Les opérations qu'il est possible de faire sur un tableau sont :

- **l'accès** à un élément du tableau

```
<tableau>[ <indice> ]
```

l'indice va de 0 à length - 1

- **la modification** d'un élément du tableau

```
<tableau>[ <indice> ] = <valeur>
```

- obtenir la **longueur** du tableau (taille **physique** du tableau)

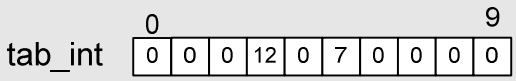
```
<tableau>.length
```

- remplacer tout le tableau par un autre.

```
<tableau> = new <type élement> [ <nouvelle taille> ]
```

Exemple :

```
int[] tab_int;

tab_int = new int[10];

tab_int[3] = 12;
tab_int[5] = 7;
```




```
int x = tab_int[3] + tab_int[5];

int n = tab_int.length;   // retourne 10

// Ecrire tous les elements du tableau
for(int i=0; i<tab_int.length;i++)
     Terminal.ecrireIntln( tab_int[i] );
```

4.2.5. Débordement d'un tableau

Si vous utilisez un indice qui est plus grand que la taille du tableau alors vous obtiendrez une erreur d'exécution :

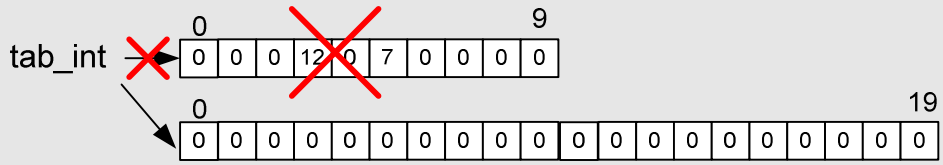
```
int y = tab_int[10];   // 10 est supérieur ou égal à tab_int.length
```

On obtient à l'exécution :

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
 at Test2.main(Test2.java:22)
```

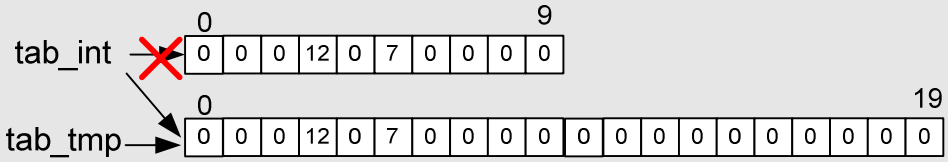
Dans le cas où on remplace tout le tableau, on perd bien sur tous les éléments du tableau.

```
tab_int = new int[20];
```



Si on veut redimensionner le tableau mais conserver les éléments du tableau, il faut faire le code suivant :

```
int tab_tmp = new int[20];

// Copie des element de tab_int dans tab_tmp
for(int i=0;i<tab_int.length;i++)
     tab_tmp[i] = tab_int[i];

// tab_int devient tab_tmp
tab_int = tab_tmp;
```



Une erreur que l'on rencontre souvent.

On oublie souvent d'allouer le tableau avant de l'utiliser.
Ce qui provoque une erreur d'exécution.
Exemple

```
int[] tab_int = null;

tab_int[0] = 12;


java Test2
Exception in thread "main" java.lang.NullPointerException
 at Test2.main(Test2.java:9)
```

4.2.6. Raccourci

Afin d'aller au plus vite, on déclare et alloue le tableau dans une même instruction :

```
int[] tab_int = new int[10];
```

4.2.7. La boucle for énumérative

La boucle for énumérative est utilisée pour parcourir les éléments d'une COLLECTION :

- tous les tableaux (primitifs ou non
- toutes les collections (qui implémente l'interface List) comme
  - **ArrayList**
  - **Vector**

Sa syntaxe est la suivante :

```
<type>[] <variable tableau>
for(<type> <variable élément> : <variable tableau> )
{
     // à chaque tour de boucle la <variable élément> prend
     //   la valeur de chacun des éléments du tableau
}
```

Attention : la variable d'itération ne peut pas être déclarée en dehors de la boucle for.

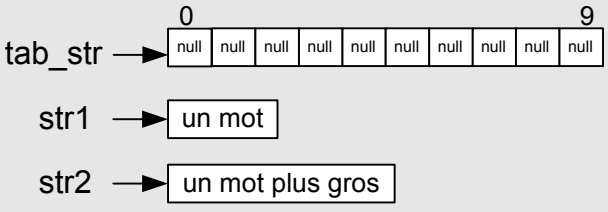
Exemple :

```
double[] tab = { 2.5, 3.0 , 4.0 };

for(double e : tab)
 {
   Terminal.ecrireStringln("élément : "+ e);
 }


java Test
element : 2.5
element : 3.0
element : 4.0
```

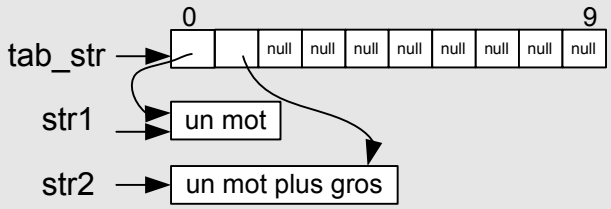
4.3. Les tableaux de String

Les tableaux de String sont en Java des tableaux d'objet.

```
String[] tab_str = new String[10];

String str1 = "un mot";
String str2 = "un mot plus gros";
```




```
tab_str[0] = str1;
tab_str[1] = str2;
```



Pour un tableau de String, faites attention à ce que chaque élément du tableau contienne bien une chaîne **allouée**.

Le code suivant est incorrect :

```
for(int i=0;i<tab_str.length;i++)
     Terminal.ecrireStringln("Longueur de "+tab_str[i]+ " est " +
                             tab_str[i].length());
```

On obtient l'exécution suivante :

```
Longueur de un mot est 6
Longueur de un mot plus gros est 16
Exception in thread "main" java.lang.NullPointerException
 at Test2.main(Test2.java:17)
```

Le bon code est :

```
for(int i=0;i<tab_str.length;i++)
        if (tab_str[i]!=null)
            Terminal.ecrireStringln("Longueur de "+tab_str[i]+ " est " +
                                    tab_str[i].length());
```

4.4. L'initialisation des tableaux

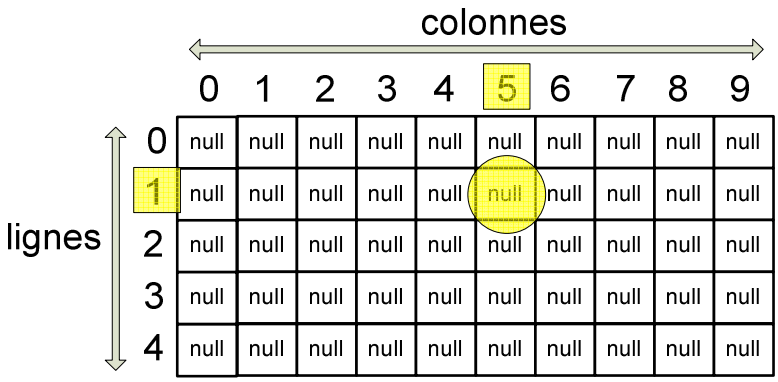
Il est possible d'initialiser les tableaux tout en allouant le tableau.

Cela est utilisé pour initialiser des tableaux de constantes ou tout simplement avec des valeurs initiales par défait.

```
int[] tab_int = {23, 34, 21, 2, 6, 10 };

String[] tab_str = { "toto", "exemple", "truc" };

String[] JOURS = { "LUNDI", "MARDI", "MERCREDI", "JEUDI", "VENDREDI",
"SAMEDI", "DIMANCHE" };
```

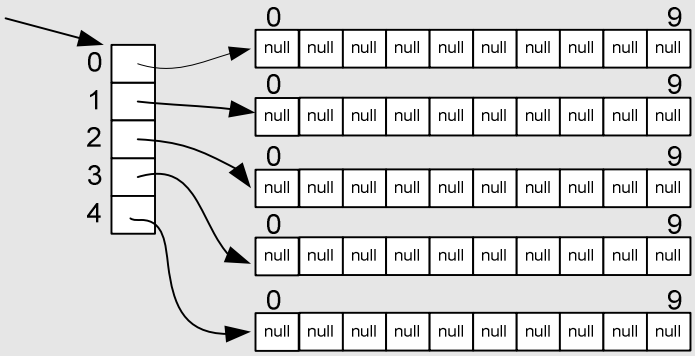
4.5. Les tableaux à 2 dimensions

Un tableau à 2 dimensions est une matrice composée de lignes et de colonnes. Chaque élément est accessible par ses coordonnées (ligne, colonne).



En Java, une matrice est un tableau dont chaque élément est un tableau.

```
String[][] matrice = new String[5][10];
```



Cette matrice se manipule de la manière suivante :

```
String[][] matrice = new String[5][10];

Terminal.ecrireStringln("Nbre de ligne   :"+matrice.length);      //5
Terminal.ecrireStringln("Nbre de colonne :"+matrice[0].length);   //10

matrice[1][5] = "toto";
```

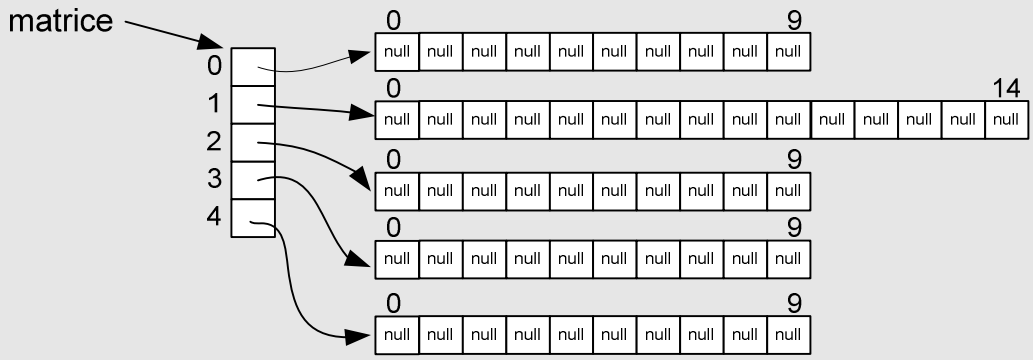
Comme pour les tableaux à 1 dimension, on peut initialiser à la déclaration une matrice :

```
int[][] tab_int = { {1, 0, 0} , {0, 1, 0} , {0, 0, 1} };
```

Les tableaux à 2 dimensions en Java ne sont pas nécessairement de même dimension :

```
          String[][] matrice = new String[5][10];

          Terminal.ecrireStringln("Nbre  de  ligne      :"+matrice.length);
//5
          Terminal.ecrireStringln("Nbre  de  colonne  :"+matrice[0].length);
//10
```

```
          matrice[1][5] = "toto";

          matrice[1] = new String[15];
```




```
          for(int i=0;i<matrice.length;i++)
              {
                  Terminal.ecrireStringln("Longueur de " + i + " : " +
matrice[i].length);
              }
          matrice[1][14] = "toto";  // Pour vérifier
```

```
  java Test2
  Nbre de ligne   :5
  Nbre de colonne :10
  Longueur de 0 : 10
  Longueur de 1 : 15
  Longueur de 2 : 10
  Longueur de 3 : 10
  Longueur de 4 : 10
```

4.6. Les tableaux à N dimensions

Même si il est difficile de représenter sur le papier ou mentalement un tableau à N dimensions, il est possible de les manipuler en informatique.

Il est assez rare de voir des tableaux au-delà de 2 dimensions.

```
  //
  int[][][][] tab; // Tableau d'entier à 4 dimensions

  tab = new int[10][20][8][12];  // Création du tableau en
                                 // donnant la dimension de chacune
                                 // des coordonnées
```

5. La classe Arrays

Afficher les tableaux : **Arrays.toString( )**

Trier les tableaux : **Arrays.sort( )**

Remplir un tableau : **Arrays.fill( )**

Egalité entre tableau : **Arrays.equals( )**

Copie entre tableau : **Arrays.copyOfRange( )**

Rechercher un elément : **Arrays.binarySearch( )**