

Cours Java

Mchaalia Raja

Plan du cours

- Introduction
- Chapitre1: Les éléments de bases en java
- Chapitre2: Classes et Objets
- Chapitre3:Héritage et Polymorphisme
- Chapitre4:Les packages et l'encapsulation
- Chapitre5: Les interfaces
- Chapitre6:Les exceptions

Introduction

Un langage Orienté Objet fortement typé avec classes

Historique:

- Le langage programmation JAVA a été développé par James Gosling chez SUNMicroSystème en 1991
- Son nom provient d'un terme d'argot désignant le café
 - Lorsque le World Wide Web a fait son apparition sur internet en 1993: ce langage s'est vu apporter quelques modifications pour s'adapter à la programmation sur le Web
- Depuis lors, il est devenu l'un des langages les plus populaires, notamment pour la programmation réseau

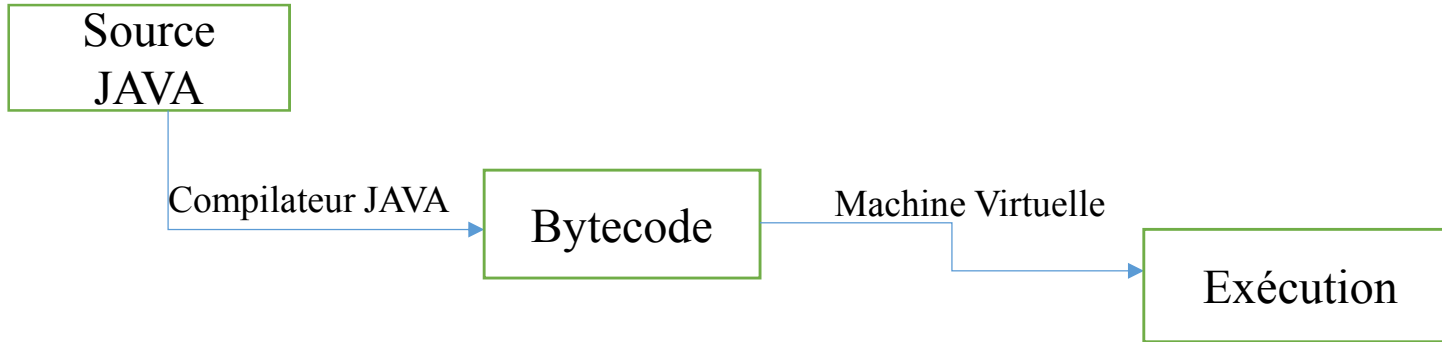
Langage Orienté Objet Multiplateforme:

- Fonctionne en mode interprété et s'exécute sur toute machine disposant de l'interpréteur disponible sur Windows, Mac, Unix, et certains mainframe
- Langage orienté objet, inspiré de C++ par certains aspects
- De nombreuses fonctionnalités Réseau et Internet
- Gestion du multitâche

Pourquoi JAVA:

- Langage fortement typé
- Langage orienté objet: Notions de classes et d'objets; Inspiré de C++
- Langage compilé vers du pseudo code binaire: Code binaire portable, nommé "ByteCode", interprété au sein d'une machine virtuelle (VM)

Portabilité = disponibilité de la machine virtuelle Java



Introduction

➤ Multitâche : le multithreading

thread = tâche spécifique et indépendante

Le multithreading est pris en charge par la machine virtuelle

Application et applet:

➤ Applications

- Exécution en dehors d'un navigateur Web
- Accès à l'ensemble des composants du système sans restriction

➤ Applets

- S'exécutent dans un navigateur Web (mode SandBox)Intégré au HTML
- Machine virtuelle intégrée au navigateur
- Applications distribuées, téléchargeables depuis un serveur HTTP
- Restrictions d'accès sur la machine locale. Exemple : ne peut dialoguer directement qu'avec la machine depuis laquelle il a été téléchargé

Outils de development:

➤ Visual Café

➤ Jbuilder

➤ Visual J++

➤ PowerJ

3environnements d'exécutions différents:

➤ JAVA ME(Micro Edition)

➤ JAVA SE(Standard Edition) pour desktop

➤ JAVA EE(Entreprise Edition)pour serveur, servlet/JSP/ JAVA mail etc...

chapitre1: Les éléments de bases en JAVA

Machine virtuelle Java:

➤ Architecture d'exécution complète

- Jeu d'instructions précis
- des registres
- une pile

➤ La sécurité est fondamentale:

- Le langage et le compilateur entièrement les pointeurs
- Un programme de vérification du bytecode veillent à l'intégrité du code java
- Le chargeur de classes (class loader) est chargé d'autoriser ou de refuser le chargement d'une classe.
- Une classe est chargée d'effectuer la vérification des appels aux API

➤ Disponibilité

- Machines virtuelles JDK
- Machines virtuelles intégrées aux navigateurs
- Machines virtuelles des environnements de développement

Java Development Kit(JDK):

Ensemble de composants permettant le développement, la mise au point et l'exécution des programmes Java.

- Un ensemble d'outils;
- Un jeu de classes et de services;
- un ensemble de spécifications.

Un développement java peut être entièrement réalisé avec le JDK, avec des outils en ligne de commande.

Les fichiers sources ont l'extension .java

Les fichiers compilés ont l'extension .class

Machine virtuelle et JDK

Nom	Description
Java.exe	Machine virtuelle java
Javac.exe	Compilateur java
Appletviewer.exe	Machine virtuelle java pour l'exécution d'applets
Jar.exe	Permet la création d'archive java
Javadoc.exe	Générateur de documentation java
Javap.exe	Désassembleur de classes compilées
jdb.exe	Débogueur en ligne de commande

Machine Virtuelle JDK

Version du JDK:

- JDK 1.0, 1.1, 1.2 (java2), 1.3 (nouvelle plate- forme java2)
 - Les JDK sont disponibles sur Internet <http://java.sun.com/products /JDK>
- JDK 1.02
 - première version réellement opérationnelle, API et classes élémentaires
 - première version de la bibliothèque AWT (Abstract Windowing Toolkit)
 - Utilisé seulement pour une compatibilité maximale.
- JDK 1.1: 1.1.8
 - améliorations et extensions du langage, améliorations de AWT
 - Apparition des composants Java et JavaBeans, JDBC (Java Database Connectivity), RMI (Remote Method Invocation)
 - Nouveau modèle d'événements, amélioration de la sécurité (signature des applets)
 - Java côté serveur (Servlets), JNI (Java Native Interface)
- JDK 1.2
 - Intégration des composants Java Swing, dans les JFC (Java Foundation Classes)
 - Amélioration des JavaBeans, intégration de l'API Java 2D, API d'accessibilité, Intégration de l'IDL en standard pour le support natif de CORBA,
 - intégration des bibliothèques CORBA
 - Support du drag and drop.

Bibliothèque:

- **AWT:** composants d'interface homme machine portables, basé sur des classes d'implémentation spécifiques à chaque plateforme. Un composant AWT correspond à un composant natif.
- **SWING :** nouvelle génération de composants; 100% Java, Look & Feel paramétrable, modèle VMC (Vue Modèle Contrôleur)

Exercice1:mon premier programme

Fichier HelloWorld .java:

```
public class HelloWorld {  
    public static void main(String[ ] args)  
    {  
        System.out.println("Hello World!");  
    }  
}
```

Les instructions Conditionnelles

- **L'instruction if:**

 If (*condition*) *instruction*;

- Exemple: afficher le plus grand entier:

```
public class Conditionnelle
{
    public static void main (String [] args)
    { int a = 33, b = 22;
      if(a > b) {
        System.out.println("a est plus grand que b"); } }}
```

- **L'instruction if.....else:** If (*condition*) *instruction1*; else *instruction2*;

- Exemple: afficher le plus grand entier

```
public class Conditionnelle
{ public static void main (String [] args)
  { int a = 33, b = 22;
    if(a > b) {
      System.out.println("a est plus grand que b"); }
    else { System.out.println("a est moins grand que b"); } } }
```

Les instructions Conditionnelles

- **L'instruction if.....else if:**

If (condition1) instruction1;

Else if(condition2)instruction2;

Else instruction3;

- Exemple: afficher le moment du jour

public class Conditionnelle1

{ public static void main (String [] args) {

String moment = "matin";

if(moment == "matin") {

System.out.println("Bonjour !");}

else if (moment == "soir") {

System.out.println("Bonsoir !"); }

else { System.out.println("Bonne nuit !"); } } }

- **L’instruction switch:**

- *Switch(var){*

Case const1: Instr1; break;

Case const2: Instr2; break;

Default: InstrN; break;}

- Exemple: afficher s’il est un homme ou une femme

public class Conditionnelle2 {

public static void main (String [] args) {

char sexe = 'H';

switch(sexe) {

case 'H':

System.out.println("Vous êtes un homme");

break;

case 'F':

System.out.println("Vous êtes une femme");

break;

default :

System.out.println("comprends pas ton sexe");

break; } } }

Les instructions répétitives

- **L'instruction while:**

- *while(expr)*

Instr;

- Exemple: calculer la somme des n premiers entiers, pour $n \geq 0$

```
public class somme{
```

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

```
int n = 10;
```

```
int som = 0; int i;
```

```
i = 1;
```

```
while( i<=n){ som = som+ i;
```

```
i = i+1; }
```

```
System.out.println(" la somme des "+n+" premiers entiers est : "+som); }}
```

- **L'instruction do.....while:**
- *do (instr)*
- *while (expr);*
- Exemple: calculer la somme des n premiers entiers, pour $n \geq 0$

```
public class somme1{  
    public static void main(String [] args){  
        int n = 10;  
        int som = 0;  
        int i=0;  
        do{ som = som+ i;  
            ++i; }  
        while(i<=n);  
        System.out.println(" la somme des "+n+" premiers entiers est : "+som);} }
```

- **L'instruction for:**
- For(expr1; expr2; expr3) instr;
- Exemple: calculer la somme des n premiers entiers, pour $n \geq 0$

```
public class somme1{  
    public static void main(String [] args){  
        int n = 10;  
        int somme = 0;  
        int i;  
        for( i=1; i<=n; ++i)  
            somme = somme + i;  
        System.out.println(" la somme des "+n+" premiers entiers est : "+somme); }}
```

- Les Boucle imbriquées:
- Exemple: Afficher des entiers compris entre 1 et 100 lu à partir du clavier:
- *public class entier{*
- *public static void main(String [] args){*
- *Int nb;*
- *boolean valide=false;*
- *do{*
- *System.out.print("Entrez un nombre entre 1 et 100:");*
- *nb=Keyboard.readInt();*
- *if((nb<1)|| (nb>100))*
- *{System.out.println("Entrée invalide");}*
- *else{*
- *valide =true;}}*
- *while(!valide);}}*

Chapitre2 :Classes et Objets

- **Définition d'une classe:** Une classe est la description de données appelées **attributs**, et d'opérations appelées **méthodes**.
- Exemple d'implémentation d'une classe en java: Classe Personne

Personne
-nom(String) -annee_naissance(int) -salaire(int)
+Personne(String, int, int) +affiche() +calcul_age()

```
import java.io.*;
class Personne{
    private String nom;
    private int annee_naissance;
    private int salaire;
    public Personne(String n, int a, int s){nom=n; annee_naissance=a; salire=s;}
    public void affiche(){System.out.println(nom+ « »+annee_naissance+« »+salaire);}
    public void calcul_age(){int age=2005-annee_naissance;System.out.println(« age =»+age);}}
```

```
public class Personne
{
    public static void main(String []args)
    {
        Personne p1=new Personne(«Dupont», 1950,1500);
        Personne p2=new Personne(«mercier», 1962,2300);
        p1.affiche();
        p2.affiche();
        p1.calcul_age();
        p2.calcul_age();}}

```

- **Les Propriétés de la classe Personne:**

Chaque propriété (ou champ) est définie par une variable, dite variable d'instance, dont on déclare le type: *String* pour le *nom* et *int* pour les deux autres variables. Chacune de ces déclarations débute par l'écriture d'un modificateur: *private*, *protected* ou *public*. Ici, *private* indique que la valeur de l'objet ne sera accessible que par la méthode de l'objet; *private* interdit donc qu'une méthode d'un objet d'une autre classe puisse directement utiliser la variable associée.

- **Le constructeur des objets de la Classe Personne:**

Toute classe doit être dotée d'une fonction particulière appelée constructeur. Ses caractéristiques sont les suivantes:

- Porter le nom de la classe.
- Être déclarée public (le modificateur public est mis en tête).
- Ne rien retourner. Ici, la méthode ***Personne(...)*** ne fait que réceptionner des valeurs et les recopier dans les variables *nom*, *annee_naissance* et *salaire*.

- **Les autres méthode de la classe `Personne`:**

La classe est dotée de deux autres méthodes: *affiche()* et *calcul_age()*. Pour chaque méthode d'une classe, il faut indiquer:

- Le niveau d'accessibilité: en général, il faut mettre `public`, ce qui signifie que la méthode peut être appelée depuis une autre classe.
- Le type de la valeur retournée.
- Le nom de la fonction et ses arguments formels avec leurs types.
- Puis le code de définition de chaque méthode. Si la fonction retourne la valeur *v*, son code se termine par *return(v)*.

- **`System.out.println(...)`:**

La méthode `println(...)` est appelée par *System.out*. Pour utiliser le code compilé de cette méthode, il faut y avoir accès. Pour cela, on doit «*importer*» la classe où figure *println*, c'est-à-dire `java.io`. Écrire *import java.io.**; permet d'accéder à l'ensemble des classes prédéfinies et compilées de *java.io*, donc en particulier la classe qui contient *println(...)*.

- **Variables et implémentation des objets:**

Le langage java est un langage fortement typé. Toute variable est donc définie par son type.

Les types des variables: Il existe deux grandes familles de types de variables: les types *ordinaires* et les types «*classe*».

- Les types ordinaires, appelés aussi élémentaires ou primitifs, sont les suivants:

- Types numériques entiers: *byte*, *short*, *int* et *long*;
- Types numériques réels: *double* et *float*;
- Type caractère: *char*;
- Type booléen: *boolean*; il est défini par deux valeurs: *false* et *true*;

- Une variable de type « *classe* » a pour valeur l'adresse de l'objet lui-même: son contenu est donc la référence à l'objet. Dans le cas de la classe *Personne*, écrire ***Personne p1***; signifie que l'on crée une variable *p1* de type « *classe* », plus précisément de type ***Personne***, et qui va ensuite avoir pour contenu l'adresse de l'objet qui sera implémenté.

- **Les trois sortes de variables:**

Pour définir une classe, il existe trois sortes de variables:

- **Les variables d'instances:** servent à définir les propriétés qui structurent une classe. Elles sont initialisées par défaut à 0 ou à nul selon leurs types.
- **Les variables de classes** sont définies par le modificateur *static*. Ces variables sont mises en place dès que la classe est chargée avant même la création des variables d'instances.
- **Les variables locales** aux méthodes sont celles que l'on définit habituellement comme variables locales aux fonctions.

- **Les variables et les objets d'un programme**

- **Les variables du programme:**

- Les variables d'instances de la classe *Personne* sont définies de la manière suivante:

{private String nom;

private int annee_naissance;

private int salaire;

- Les trois variables sont indiquées par le modificateur *private* et avec les types *String* et *int*. Elles structurent chaque objet, instance de la classe *Personne*.

- La création des objets désignés par *p1* et *p2*

Personne *p1*

p1=new Personne(« Dupont», 1950,1500)

Exécuter *Personne p1* revient à mettre une variable de nom *p1* qui est de type « classe ». Elle est initialisée par défaut, sa valeur vaut nul. Puis l'instruction *new* est exécutée

- *new* est une instruction du langage java qui procède toujours de la manière suivante:

1. Alloue la mémoire de l'objet
2. Appelle la méthode constructeur et exécute le constructeur
3. Retourne l'adresse de l'objet ainsi construit.

- Pour cet exemple, le déroulement est le suivant:

- ***Allocation mémoire de l'objet lui-même:*** l'allocation dynamique se fait en fonction des types des variables qui structurent l'ensemble des propriétés de l'objet. Elle se fait selon la définition des variables *nom annee_naiss* et *salaire*. Une initialisation par défaut se produit: *nom= null; annee_naiss=0; salaire=0*.
- ***Appel du constructeur, puis exécution de la méthode correspondante:*** Ici, appel de la méthode *Personne(« Dupont», 1950, 1500)* puis exécution. Cette méthode réceptionne les valeurs des paramètres et les recopie dans les variables de l'objet.
- ***new retourne l'adresse de l'objet ainsi construit:*** Cette adresse est mise *p1*. le processus est le même pour la *personne p2*.

	nom	Dupont	
Annee_naissance		1950	
salaire		1500	
	nom	Mercier	
Annee_naissance		1962	
salaire		2300	
p2	Adresse		
p1	Adresse		

- **Le statut des variables p1 et p2**

Les variables *p1* et *p2* au sein de la fonction main: elles ont donc le statut des variables locales à cette fonction. Elles prennent place en mémoire dès qu'elles sont créées. Mais, dès la fin de l'exécution de la fonction main, les variables *p1* et *p2* sont détruites. À cet instant, les variables des objets *nom*, *annee_naissance* et *salaire* ne sont plus référencées: L'environnement java assure alors la destruction.

- **Les méthodes: les accesseurs:**

Deux types de méthodes servent à donner accès aux variables depuis l'extérieur de la classe :

➤ Les accesseurs en lecture pour lire les valeurs des variables ; « *accesseur en lecture* » est souvent abrégé en « *accesseur* » ; *getter* en anglais.

➤ Les accesseurs en écriture, ou modificateurs, pour modifier leur valeur ; *setter* en anglais.

- Exemple: `void set_salaire(){int s1; salaire=s1;} int get_salaire(){return s1;}`

- **This:** Le code d'une méthode d'instance désigne l'instance qui a reçu le message (l'instance courante), par le mot-clé *this*, donc, les membres de l'instance courante en les préfixant par «*this.*»

- Lorsqu'il n'y a pas d'ambiguïté, *this* est optionnel pour désigner un membre de l'instance courante.

- **toString():** Il est conseillé d'inclure une méthode **toString** dans toutes les classes que l'on écrit

- Cette méthode renvoie une chaîne de caractères qui décrit l'instance

- Cette description peut être très utile lors de la mise au point des programmes

- Cette description doit être compacte et précise.

- **System.out.println(objet)** affiche la valeur retournée par **objet.toString()**

- Exemple:

```
public class Personne{.....
```

```
public String toString(){return « nom: » +nom+ « \n annee_naiss: » +annee_naiss+ .....;}
```

- **Le ramasse-miettes**(en anglais Garbage Collector) Son principe est le suivant :

- À tout instant, on connaît le nombre de références à un objet donné. On notera que cela n'est possible que parce que Java gère toujours un objet par référence.
- Lorsqu'il n'existe plus aucune référence sur un objet, on est certain que le programme ne pourra plus y accéder. Il est donc possible de libérer l'emplacement correspondant, qui pourra être utilisé pour autre chose. Cependant, pour des questions d'efficacité, Java n'impose pas que ce travail de récupération se fasse immédiatement. En fait, on dit que l'objet est devenu candidat au ramasse-miettes.

- Exemple :

```
public class Circle {...
```

```
void finalize() { System.out.println("Je suis garbage collector"); }}
```

...

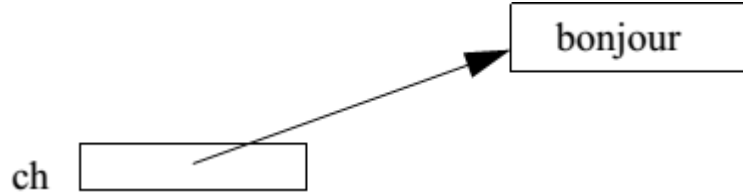
```
Circle c1;
```

```
if (condition) {Circle c2 = new Circle(); // c2 référence une nouvelle instance
```

```
c1 = c2;}// La référence c2 n'est plus valide mais il reste une référence,c1, sur l'instance c1=null; // L'instance ne possède plus de référence. Elle n'est plus // accessible. A tout moment le gc peut détruire l'objet.
```


Les chaînes de caractères

- **Introduction:** Comme toute déclaration d'une variable objet, l'instruction : *String ch* ; déclare que *ch* est destinée à contenir une référence à un objet de type *String*.



- La *classe String* dispose de deux constructeurs, l'un sans argument créant une chaîne vide, l'autre avec un argument de type *String* qui en crée une copie :

String ch1 = new String () ; // ch1 contient la référence à une chaîne vide

String ch2 = new String("hello") ; // ch2 contient la référence à une chaîne contenant la suite "hello"

String ch3 = new String(ch2) ; // ch3 contient la référence à une chaîne copie de ch2, donc contenant "hello"

- **Entrées sorties de chaîne:** Nous avons déjà vu qu'on pouvait afficher des constantes chaînes par la méthode *println*:

System.out.println ("bonjour") ;

En fait, cette méthode reçoit la référence à une chaîne. Elle peut donc aussi être utilisée de cette manière :

String ch ;

.....System.out.println (ch) ;

Par ailleurs, comme pour les types primitifs ou les autres types classes, il n'existe pas de méthode standard permettant de lire une chaîne au clavier. C'est pourquoi nous avons doté notre classe utilitaire *Clavier* d'une méthode (statique) nommée *lireString*,

- Voici comment vous pourrez lire une chaîne de caractères quelconques fournis au clavier et obtenir sa référence dans `ch` (la méthode crée automatiquement l'objet de type `String` nécessaire) :

String ch ;

.....

ch = Clavier.lireString() ; // crée un objet de type String contenant la// référence à une chaîne lue au clavier

- **Longueur d'une chaîne : length:** La méthode *length* permet d'obtenir la longueur d'une chaîne:

String ch = "bonjour" ;

int n = ch.length() ; // n contient 7

ch = "hello" ; n = ch.length () ; // n contient 5

ch = "" ; n = ch.length () ; // n contient 0

- **Accès au caractères d'une chaîne: charAt:** La méthode `charAt` de la classe `String` permet d'accéder à un caractère de rang donné d'une chaîne (le premier caractère porte le rang 0). Ainsi, avec :

String ch = "bonjour" ;

ch.charAt(0)correspond au caractère 'b',

ch.charAt(2)correspond au caractère 'n'.

Voici un exemple d'un programme qui lit une chaîne au clavier et l'affiche verticalement, c'est-à-dire à raison d'un caractère par ligne :

```
public class MotCol{ public static void main (String args[]){ String mot ;System.out.print ("donnez un mot : ") ;  
mot = Clavier.lireString() ;  
System.out.println ("voici votre mot en colonne :") ;  
for (int i=0 ; i<mot.length() ; i++)System.out.println (mot.charAt(i)) ;}}
```

donnez un mot : Langage voici votre mot en colonne :

L

a

n

g

a

g

e

- **Concaténation des chaines:** L'opérateur + est défini lorsque ses deux opérandes sont des chaînes. Il fournit en résultat une nouvelle chaîne formée de la concaténation des deux autres, c'est-à-dire contenant successivement les caractères de son premier opérande, suivis de ceux de son second.

```
String ch1 = "Le langage " ;
```

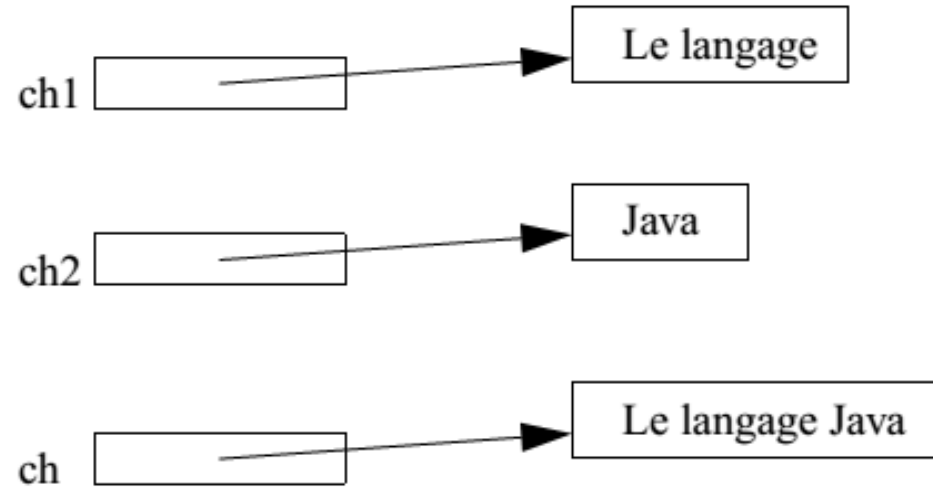
```
String ch2 = "Java" ;
```

```
String ch = ch1 + ch2 ;
```

- Dorénavant, il existe trois objets de type String dont les références sont dans ch1, ch2 et ch. Bien entendu, dans une instruction telle que :

System.out.println (ch1 + ch2) ;

- l'évaluation de l'expression ch1+ch2 crée un nouvel objet de type String et y place la chaîne



- **Recherche dans une chaîne:** La méthode *indexOf* surdéfinie dans la classe *String* permet de rechercher, à partir du début d'une chaîne ou d'une position donnée :
- la première occurrence d'un caractère donné,
- la première occurrence d'une autre chaîne.

Dans tous les cas, elle fournit :

- la position du caractère (ou du début de la chaîne recherchée) si une correspondance a effectivement été trouvée,
- la valeur -1 sinon.

- Il existe également une méthode ***lastIndexOf***, surdéfinie pour effectuer les mêmes recherches que `indexOf`, mais en examinant la chaîne depuis sa fin. Voyez ces instructions :

```
String mot = "anticonstitutionnellement" ;
```

```
int n ;
```

```
n = mot.indexOf('t') ; // n vaut 2
```

```
n = mot.lastIndexOf('t') ; // n vaut 24
```

```
n = mot.indexOf("ti") ; // n vaut 2
```

```
n = mot.lastIndexOf("ti") ; // n vaut 12
```

```
n = mot.indexOf('x') ; // n vaut -1
```

- **Extraction des sous-chaînes:** La méthode *substring* permet de créer une nouvelle chaîne en extrayant de la chaîne courante

➤ soit tous les caractères depuis une position donnée,

```
String ch = "anticonstitutionnellement" ;
```

```
String ch1 = ch.substring(5) ; // ch n'est pas modifiée // ch1 contient "onstitutionnellement"
```

➤ soit tous les caractères compris entre deux positions données (la première incluse, la seconde exclue) :

```
String ch = "anticonstitutionnellement" ;
```

```
String ch1 = ch.substring(4, 16) ; // ch n'est pas modifiée // ch1 contient "constitution"
```

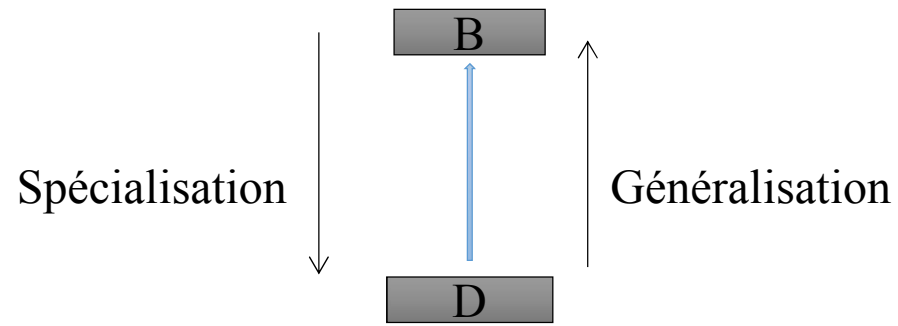
- **La Classe *StringBuffer***: Les objets de type *String* ne sont pas modifiables mais nous avons vu qu'il était possible de les employer pour effectuer la plupart des manipulations de chaînes. Cependant, la moindre modification d'une chaîne ne peut se faire qu'en créant une nouvelle chaîne (c'est le cas d'une simple concaténation). Dans les programmes manipulant intensivement des chaînes, la perte de temps qui en résulte peut devenir gênante. C'est pourquoi Java dispose d'une classe ***StringBuffer*** destinée elle aussi à la manipulation de chaînes, mais dans laquelle les objets sont modifiables.
- La classe ***StringBuffer*** dispose de fonctionnalités classiques mais elle n'a aucun lien d'héritage avec *String* et ses méthodes ne portent pas toujours le même nom. On peut créer un objet de type ***StringBuffer*** à partir d'un objet de type *String*. Il existe des méthodes :
 - de modification d'un caractère de rang donné : *setCharAt*,
 - d'accès à un caractère de rang donné : *charAt*,
 - d'ajout d'une chaîne en fin : la méthode *append* accepte des arguments de tout type primitif et de type *String*,
 - d'insertion d'une chaîne en un emplacement donné : *insert*,
 - de remplacement d'une partie par une chaîne donnée : *replace*,
 - de conversion de *StringBuffer* en *String*: *toString*.

```
class TstStB  
{ public static void main (String args[])  
{ String ch = "la java" ;  
StringBuffer chBuf = new StringBuffer (ch) ;  
System.out.println (chBuf) ;  
chBuf.setCharAt (3, 'J'); System.out.println (chBuf) ;  
chBuf.setCharAt (1, 'e') ; System.out.println (chBuf) ;  
chBuf.append (" 2") ; System.out.println (chBuf) ;  
chBuf.insert (3, "langage ") ; System.out.println (chBuf) ;}}
```

- la java
- la Java
- le Java
- le Java 2
- le langage Java 2

Héritage et Polymorphisme

- **Héritage:**
- **Définition:** Une classe Java dérive toujours d'une autre classe, `Object` quand rien n'est spécifié. La classe dérivée possède les propriétés suivantes:
 - contient les données membres de la classe de base,
 - peut en posséder de nouvelles,
 - possède (à priori) les méthodes de sa classe de base,
 - peut redéfinir (masquer) certaines méthodes,
 - peut posséder de nouvelles méthodes.
- **Syntaxe:**
- Pour spécifier de quelle classe hérite une classe on utilise le mot-clé *extends* :
- *class D extends B { . . . }*
- La classe D dérive de la classe B. On dit que la classe B est la super classe, la classe de base, ou la classe mère de la classe dérivée D, et que D dérive de B, ou que D est une sous-classe de B.



- La visibilité *protected* rend l'accès possible :
- Depuis les classes dérivée.
- Depuis les classes du paquetage.

Exemple : `class B { private int a; protected int b; int c; public int d; }`

- **This et super:** Chaque instance est munie de deux références particulières :
- **this** réfère l'instance elle-même.
- **super** réfère la partie héritée de l'instance.
- Exemple : `class D extends B { ...`

`D (private int e;){super(b,c);} }`

Polymorphisme

- Soit la classe suivante:

```
import java.io.*;

class point{protected int x, y;

public point(int a,int b){x=a; y=b;}

public void modif(int coef){x=x*coef; y=y*coef;}

public void affiche_pt(){System.out.println (« coordonné du point: » +x+ « » +y );}}

class point_colore extends point{private String couleur;

public point_colore(int a, int b, String c){super(a,b); couleur=c;}

public void affiche_pt(){super.affiche_pt();

System.out.println (« et sa couleur: » +couleur);}}

.....

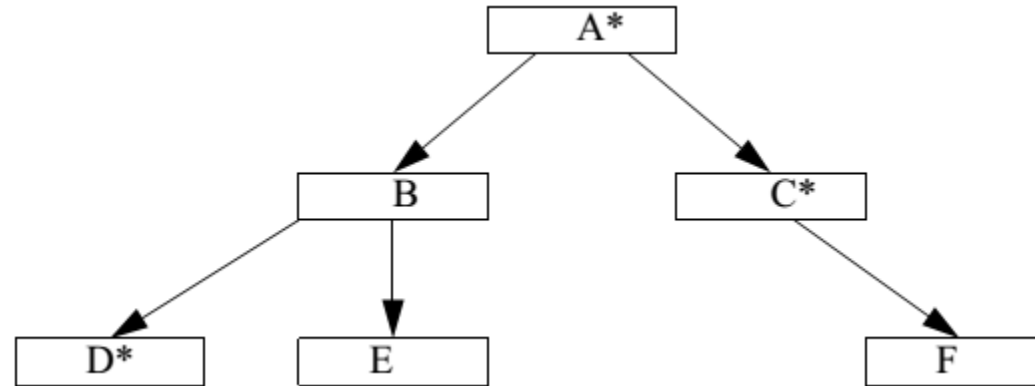
class point_physique extends point{private double point;

public point_physique(int a, int b, double p){super(a,b); point=b;}}

.....
```

- **Généralisation à plusieurs classes**

Nous venons de vous exposer les fondements du polymorphisme en ne considérant que deux classes. Mais il va de soi qu'ils se généralisent à une hiérarchie quelconque. Considérons à nouveau la hiérarchie de classes présentée par la figure suivante, dans laquelle seules les classes marquées d'un astérisque définissent ou redéfinissent la méthode f:



Avec ces déclarations :

A a ; B b ; C c ; D d ; E e ; F f ;

les affectations suivantes sont légales :

a = b ; a = c ; a = d ; a = e ; a = f ; b = d ; b = e ; c = f ;

En revanche, celles-ci ne le sont pas :

b = a ; // erreur : A ne descend pas de B

d = c ; // erreur : C ne descend pas de D

c = d ; // erreur : D ne descend pas de C

Voici quelques exemples précisant la méthode f appelée, selon la nature de l'objet effectivement référencé par a(de type A) :

a référence un objet de type A : méthode f de A

a référence un objet de type B : méthode f de A

a référence un objet de type C : méthode f de C

a référence un objet de type D : méthode f de D

a référence un objet de type E : méthode f de A

a référence un objet de type F : méthode f de C.

- **Polymorphisme, redéfinition et sur-définition**

Par essence, le polymorphisme se fonde sur la redéfinition des méthodes. Mais il est aussi possible de sur-définir une méthode. Cependant, nous n'avions pas tenu compte alors des possibilités de polymorphisme qui peuvent conduire à des situations assez complexes. En voici un exemple :

```
class A
{ public void f (float x) { ..... }
.....}

class B extends A
{ public void f (float x) { ..... } // redéfinition de f de A
public void f (int n) { ..... } // sur-définition de f pour A et B
.....}

A a = new A(...) ;

B b = new B(...) ; int n ;

a.f(n) ; // appelle f (float) de A (ce qui est logique)
b.f(n) ; // appelle f(int) de B comme on s'y attend

a = b ; // a contient une référence sur un objet de type B

a.f(n) ; // appelle f(float) de B et non f(int)
```

Les Paquetages

- La notion de paquetage ou packages correspond à un regroupement logique sous un identificateur commun d'un ensemble de classes. Elle est proche de la notion de bibliothèque que l'on rencontre dans d'autres langages. Elle facilite le développement et la cohabitation de logiciels conséquents en permettant de répartir les classes correspondantes dans différents paquetages.
- Un package est un ensemble de classes ayant son propre identificateur. Tout programmeur peut définir ses propres packages en indiquant leurs noms au début du programme.
- **Attribution d'une classe à un paquetage**

Un paquetage est caractérisé par un nom qui est soit un simple identificateur, soit une suite d'identificateurs séparés par des points, comme dans :

MesClasses

Utilitaires.Mathematiques

Utilitaires.Tris

L'attribution d'un nom de paquetage se fait au niveau du fichier source ; toutes les classes d'un même fichier source appartiendront donc toujours à un même paquetage. Pour ce faire, on place, en début de fichier, une instruction de la forme :

package xxxxxx ;

Dans laquelle xxxxxx représente le nom du paquetage.

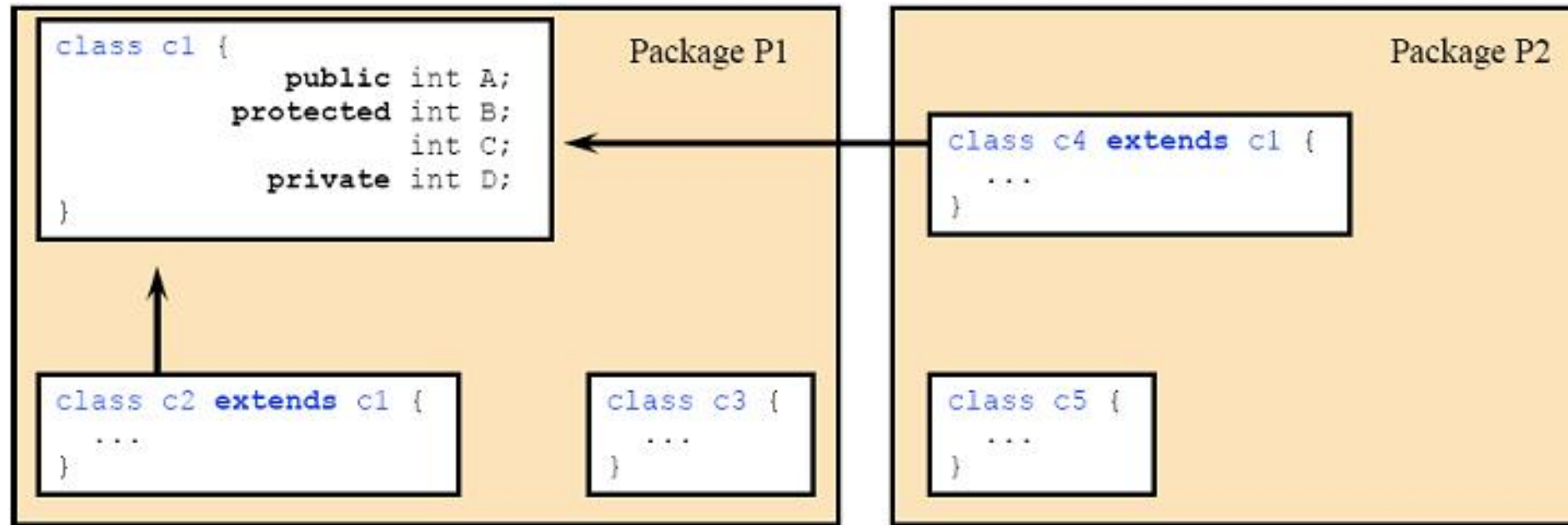
- **Utilisation d'une classe d'un paquetage**

Lorsque, dans un programme, vous faites référence à une classe, le compilateur la recherche dans le paquetage par défaut. Pour utiliser une classe appartenant à un autre paquetage, il est nécessaire de fournir l'information correspondante au compilateur. Pour ce faire, vous pouvez :

- citer le nom du paquetage avec le nom de la classe,
- utiliser une instruction import en y citant soit une classe particulière d'un paquetage, soit tout un paquetage.

- **En citant le nom de la classe**
- Si vous avez attribué à la classe **Point** le nom de paquetage *MesClasses* par exemple, vous pourrez l'utiliser simplement en la nommant *MesClasses.Point*. Par exemple :
- *MesClasses.Point p = new MesClasses.Point (2, 5) ;*
-
- *p.affiche()* ; // ici, le nom de paquetage n'est pas requis
- Evidemment, cette démarche devient fastidieuse dès que de nombreuses classes sont concernées.
- **En important une classe**
- L'instruction *import* vous permet de citer le nom (complet) d'une ou plusieurs classes, par exemple :
- *import MesClasses.Point, MesClasses.Cercle ;*
- À partir de là, vous pourrez utiliser les classes *Point* et *Cercle* sans avoir à mentionner leur nom de paquetage, comme si elles appartenaient au paquetage par défaut.
- **En important un paquetage**
- La démarche précédente s'avère elle aussi fastidieuse dès qu'un certain nombre de classes d'un même paquetage sont concernées. Avec :
- *import MesClasses.* ;*

L'encapsulation



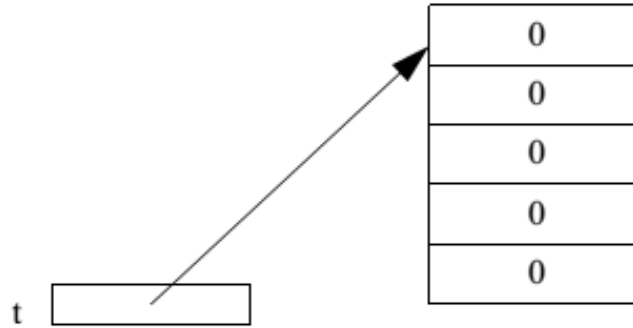
	A	B	C	D
Accessible par c2	o	o	o	-
Accessible par c3	o	o	o	-
Accessible par c4	o	o	-	-
Accessible par c5	o	-	-	-

- **Définition:** Le terme encapsulation désigne le principe consistant à cacher l'information contenue dans un objet et de ne proposer que des méthodes de modification/accès à ces propriétés (attributs).
 - L'objet est vu de l'extérieur comme une boîte noire ayant certaines propriétés et ayant un comportement spécifié.
 - La manière dont le comportement a été implémenté est cachée aux utilisateurs de l'objet.
- **Intérêt:** Protéger la structure interne de l'objet contre toute manipulation non contrôlée, produisant une incohérence.
- **Encapsulation en pratique:** L'encapsulation nécessite la spécification de parties publics et privées de l'objet.
 - ✓ éléments publics : correspond à la partie visible de l'objet depuis l'extérieur. c'est un ensemble de méthodes utilisables par d'autres objets (environnement).
 - ✓ éléments privées : correspond à la partie non visible de l'objet. Il est constitué des éléments de l'objet visibles uniquement de l'intérieur de l'objet et de la définition des méthodes.

Les tableaux: déclaration et création des tableaux

`t = new int[5] ; // t fait référence à un tableau de 5 entiers`

- Cette instruction alloue l'emplacement nécessaire à un tableau de 5 éléments de type int et en place la référence dans t. Les 5 éléments sont initialisés par défaut (comme tous les champs d'un objet) à une valeur "nulle" (0 pour un int). On peut illustrer la situation par ce schéma :



- **Exemple d'utilisation de Scanner:**


```
import java.util.Scanner;  
  
public class Main {  
    public static void main (String[] args) {  
        int age;  
  
        Scanner sc = new Scanner (System.in);  
  
        System.out.print("Saisissez votre âge : ");  
  
        age = sc.nextInt();  
  
        if (age > 20)  
            System.out.println("vous êtes âgé");  
        else if (age > 0)  
            System.out.println("vous êtes jeune");  
        else  
            System.out.println("vous êtes en devenir");  
    }  
}
```

La gestion des exceptions

- **Introduction:** Il est fréquent que le traitement d'une anomalie ne puisse pas être fait par la méthode l'ayant détectée, mais seulement par une méthode ayant provoqué son appel. Cette dissociation entre la détection d'une anomalie et son traitement peut obliger le concepteur à utiliser des valeurs de retour de méthode servant de "compte rendu". Là encore, le programme peut très vite devenir complexe ; de plus, la démarche ne peut pas s'appliquer à des méthodes sans valeur de retour donc, en particulier, aux constructeurs. La situation peut encore empirer lorsque l'on développe des classes réutilisables destinées à être exploitées par de nombreux programmes. Java dispose d'un mécanisme très souple nommé *gestion d'exception*, qui permet à la fois :

- de dissocier la détection d'une anomalie de son traitement,
- de séparer la gestion des anomalies du reste du code, donc de contribuer à la lisibilité des programmes.

D'une manière générale, une exception est une rupture de séquence déclenchée par une instruction *throw* comportant une expression de type classe. Il y a alors branchement à un ensemble d'instructions nommé "gestionnaire d'exception". Le choix du bon gestionnaire est fait en fonction du type de l'objet mentionné à *throw* (de façon comparable au choix d'une fonction surdéfinie).

- **Comment déclencher une exception avec *throw***  Considérons une classe Point, munie d'un constructeur à deux arguments et d'une méthode affiche. Supposons que l'on ne souhaite manipuler que des points ayant des coordonnées non négatives. Nous pouvons, au sein du constructeur, vérifier la validité des paramètres fournis. Lorsque l'un d'entre eux est incorrect, nous "déclenchons" une exception à l'aide de l'instruction *throw*. À celle-ci, nous devons fournir un objet dont le type servira ultérieurement à identifier l'exception concernée. Nous créons donc (un peu artificiellement) une classe que nous nommerons *ErrCoord*. Java impose que cette classe dérive de la classe standard Exception. Pour l'instant, nous n'y plaçons aucun membre (mais nous le ferons dans d'autres exemples) :

class ErrConst extends Exception { }

- Pour lancer une exception de ce type au sein de notre constructeur, nous fournirons à l'instruction throw un objet de type ErrConst, par exemple de cette façon :

throw new ErrConst() ;

- En définitive, le constructeur de notre classe Point peut se présenter ainsi :

class Point

{ public Point(int x, int y) throws ErrConst

{ if ((x<0) || (y<0)) throw new ErrConst() ; // lance une exception de type ErrConst

this.x = x ; this.y = y ;}

- Notez la présence de throws ErrConst, dans l'en-tête du constructeur, qui précise que la méthode est susceptible de déclencher une exception de type ErrConst. Cette indication est obligatoire en Java, à partir du moment où l'exception en question n'est pas traitée par la méthode elle-même.
- En résumé, voici la définition complète de nos classes Point et ErrCoord:

```
class Point  
{private int x, y ;  
public Point(int x, int y) throws ErrConst  
{ if ( (x<0) || (y<0)) throw new ErrConst() ;  
this.x = x ; this.y = y ;}  
public void affiche()  
{ System.out.println ("coordonnees : " + x + " " + y) ;}}  
class ErrConst extends Exception { }
```

- **Utilisation d'un gestionnaire d'exception:** Disposant de notre classe `Point`, voyons maintenant comment procéder pour gérer convenablement les éventuelles exceptions de type *ErrConst* que son emploi peut déclencher. Pour ce faire, il faut :
 - inclure dans un bloc particulier dit "bloc **try**" les instructions dans lesquelles on risque de voir déclenchée une telle exception ; un tel bloc se présente ainsi :

```
try{// instructions}
```

- faire suivre ce bloc de la définition des différents gestionnaires d'exception (ici, un seul suffit). Chaque définition de gestionnaire est précédée d'un en-tête introduit par le mot-clé **catch** (comme si `catch` était le nom d'une méthode gestionnaire). Voici ce que pourrait être notre unique gestionnaire :

```
catch (ErrConst e) { System.out.println ("Erreur construction ") ; System.exit (-1) ;}
```

- Ici, il se contente d'afficher un message et d'interrompre l'exécution du programme en appelant la méthode standard *System.exit* (la valeur de l'argument est transmis à l'environnement qui peut éventuellement l'utiliser comme "compte rendu").


```

class Point {private int x, y ;
public Point(int x, int y) throws ErrConst
{ if ( (x<0) || (y<0)) throw new ErrConst() ;
this.x = x ; this.y = y ;}
public void affiche()
{ System.out.println ("coordonnees : " + x + " " + y) ;}}

class ErrConst extends Exception{ }

public class Except1
{ public static void main (String args [])
{ try{ Point a = new Point (1, 4) ;
a.affiche() ;
a = new Point (-3, 5) ;
a.affiche() ;}
catch (ErrConst e){ System.out.println ("Erreur construction ") ;
System.exit (-1) ;}}}

coordonnees : 1 4
Erreur construction

```

- **Exercice: chaîne de caractère:** Écrire un programme qui lit un mot au clavier et qui indique combien de fois sont présentes chacune des voyelles a, e, i, o, u ou y, que celles-ci soient écrites en majuscules ou en minuscules, comme dans cet exemple : donnez un mot : Anticonstitutionnellement il comporte
 - 1 fois la lettre a
 - 3 fois la lettre e
 - 3 fois la lettre i
 - 2 fois la lettre o
 - 1 fois la lettre u
 - 0 fois la lettre y

- Corrigé:

```
public class Voyelles{ public static void main (String args[])  
{ char voy[] = {'a', 'e', 'i', 'o', 'u', 'y'} ;  
int nVoy [] = new int [voy.length] ;  
for (int i=0 ; i<nVoy.length ; i++) nVoy[i] = 0 ;  
System.out.print ("donnez un mot : ") ;  
String mot = Clavier.lireString() ;  
mot = mot.toLowerCase() ;  
for (int i=0 ; i<mot.length() ; i++)  
for (int j=0 ; j<voy.length ; j++)  
if (mot.charAt(i) == voy[j]) nVoy[j]++ ;  
System.out.println ("il comporte : ") ;  
for (int i=0 ; i<voy.length ; i++) System.out.println(nVoy[i] + " fois la lettre " + voy[i]) ; }}
```

L'instruction : `mot = mot.toLowerCase() ;`

crée une nouvelle chaîne obtenue par conversion en minuscules de la chaîne référencée par `mot`, puis place son adresse dans `mot`. Il n'y a pas modification de la chaîne initiale. Ici, toutefois, celle-ci devenant non référencée, deviendra candidate au ramasse-miettes...

- **Exercice: tableau:**Écrire un programme qui :
- lit dans un tableau 5 valeurs flottantes fournies au clavier
- on calcule et on affiche la moyenne, la plus grande et la plus petite valeur.

Corrigé: *public class UtilTab1{ public static void main (String args[])*

{ final int N = 5 ;

double val [] = new double[N] ;int i ;

System.out.println ("donnez " + N + " valeurs flottantes") ;

for (i=0 ; i<N ; i++) // for... each n'est pas applicable ici

val[i] = Clavier.lireDouble() ;

double valMax = val[0], valMin = val[0], somme=0 ;

for (i=0 ; i<N ; i++) { if (val[i] > valMax) valMax = val[i] ; // for (double v : val)

if (val[i] < valMin) valMin = val[i] ; // { if (v>valMax) valMax=v ; // if (v<valMin) valMin=v ; // som += v ;

somme += val[i] ; } }System.out.println ("valeur maximale = " + valMax) ;

System.out.println ("valeur minimale = " + valMin) ;

double vMoyenne = somme/N ; // on suppose que N est strictement positif

System.out.println ("moyenne " + vMoyenne) ;}}