Chapitre 5
Héritage

Introduction

Comme pour les pour autres langages orientés objets, Java permet la notion d'héritage, qui permet de créer de nouvelles classes à partir d'autres classes existantes. L'héritage permet de réutiliser des classes déjà définies en adaptant les attributs et les méthodes (par ajout et/ou par modification).

Une classe qui hérite d'une classe existante est appelée classe **dérivée**. Elle est aussi appelée **sous-classe** ou **classe-fille**.

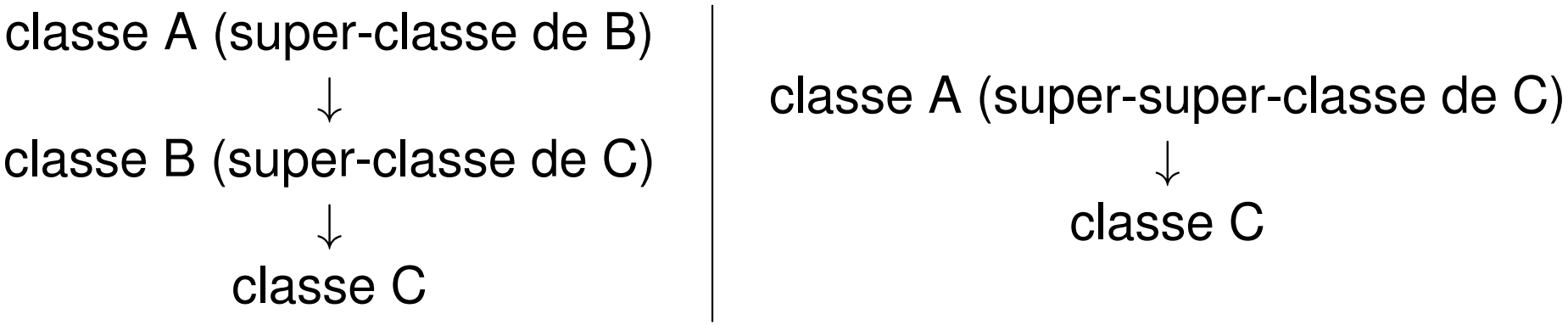
La classe, dont hérite d'autres classes, est appelée classe **super-classe**. Elle est aussi appelée **classe-mère** ou **classe-parente**.

Syntaxe

```
class SousClasse extends SuperClass
```

Remarques :

- Java ne permet pas l'héritage multiple. C'est-à-dire, une classe ne peut pas hériter de plusieurs classes. Elle ne peut hériter que d'une seule classe.
- Une classe peut hériter d'une classe dérivée. Considérons la classe **A** qui est la **super-classe** de **B** et **B** qui est la **super-classe** de **C**.
  $\longrightarrow$ **A** est la **super-super-classe** de **C**.



Exemple introductif

Considérons les deux classes : **Etudiant** et **Professeur**. Pour les deux classes :

1. les attributs **nom** et **prenom** sont en commun ;
2. les méthodes **afficher()** et **setNom()** sont en commun ;
3. la classe **Etudiant** contient l'attribut **cne** et la classe **Professeur** contient l'attribut **cin**.

Exemple : Classe Etudiant

```
class Etudiant {
    private String nom, prenom, cne;

    void afficher() {
        System.out.println("Nom :"+nom);
        System.out.println("Prenom :"+prenom);
    }
    void setNom(String nom){
        this.nom = nom;
    }
}
```

Exemple : Classe Professeur

```
class Professeur {
    private String nom, prenom, cin;

    void afficher() {
        System.out.println("Nom :"+nom);
        System.out.println("Prenom :"+prenom);
    }
    void setNom(String nom){
        this.nom = nom;
    }
    void setCin(String cin){
        this.cin = cin;
    }
}
```

Utilisation de l'héritage

Un étudiant et un professeur sont des personnes. Définissons une nouvelle classe **Personne** :

```
class Personne {
    private String nom, prenom;

    void afficher() {
        System.out.println("Nom :"+nom);
        System.out.println("Prenom :"+
            prenom);
    }
    void setNom(String nom){
        this.nom = nom;
    }
}
```

Les deux classes peuvent être modifiées en utilisant la classe **Personne** . Elle deviennent comme le montre le listing suivant :

```
class Etudiant extends Personne {
    private String cne;
    void setCne(String cne){
        this.cne = cne;
    }
}

class Professeur extends Personne{
    private String cin;

    void setCin(String cin){
        this.cin = cin;
    }
}
```

Accès aux attributs

L'accès aux attributs privés (**private**) d'une super-classe n'est pas permis de façon directe. Supposons qu'on veut définir, dans classe **Etudiant**, une méthode **getNom()** qui retourne le nom alors, l'instruction suivante n'est pas permise puisque le champ **Personne.nom** est non visible (`The field Personne.nom is not visible`).

```
class Etudiant extends Personne {
    private String cne;
    String getNom() {
        return nom; //non permise
    }
}
```

Accès aux attributs

Pour accéder à un attribut d'une **super-classe**, il faut soit :

- rendre l'attribut publique, ce qui implique que ce dernier est accessible par toutes les autres classes ;
- définir, dans la classe **Personne**, des méthodes qui permettent d'accéder aux attributs privés (**getters** et **setters**) ;
- déclarer l'attribut comme protégé en utilisant le mot clés **protected**.

Exemple

La classe **Etudiant** peut accéder à l'attribut **nom** puisqu'il est protégé.

```
class Personne {
    protected String nom;
    ...
}
class Etudiant extends Personne {
    ...
    String getNom() {
        return nom;
    }
}
```

Remarques :

1. Un attribut protégé est accessible par toutes les sous-classes et par toutes les classes du même paquetage (on verra plus loin la notion de **package**) ce qui casse l'encapsulation.
2. Le mode protégé n'est pas très utilisé en Java.

Héritage hiérarchique

Comme mentionné dans l'introduction, une classe peut être la **super-super-classe** d'une autre classe. Reprenons l'exemple concernant l'héritage et ajoutons la classe **EtudiantEtranger**. Un étudiant étranger est lui aussi un étudiant dont on veut lui ajouter la nationalité .

```
class Personne {
  ...
}
class Etudiant extends Personne {
    ...
}
class EtudiantEtranger extends Etudiant {
    private String nationalite;
    ...
}
```

Définitions

Masquage (shadowing)

un attribut d'une sous-classe qui porte le même nom qu'un autre attribut de la super-classe.

Peu utilisé en pratique par-ce-qu'il est source d'Ambiguïté .

Redéfinition (overriding)

comme pour le cas de surcharge à l'intérieure d'une classe, une méthode déjà définie dans une super-classe peut avoir une nouvelle définition dans une sous-classe.

Remarques :

1. Il ne faut pas confondre surcharge et redéfinition !
2. On verra plus de détails concernant la redéfinition dans le chapitre concernant le polymorphisme.

Exemple : masquage

```
public class Masquage {
    public static void main(String[] args){
        Masquage masq = new Masquage();
        masq.affiche();   //Affichera 10
        masq.variableLocal();//Affichera 20
    }
    private int a = 10;
    public void affiche()
    {
        System.out.println("a = " + a);
    }
    public void variableLocal()
    {
        int a = 20; //variable locale
        System.out.println("a = " + a);
    }
}
```

Redéfinition : Exemple 1

```
class A
{
   public void f(int a, int b)
   {
      //instructions
   }
   //Autres methodes et attributs
}
class B extends A
{
   public void f(int a, int b)
   {
      //la methode redifinie f() de la
          super-classe
   }
   //Autres methodes et attributs
```

Redéfinition : Exemple 2

Reprenons la classe **Personne** et ajoutons à cette classe la méthode **afficher()** qui permet d'afficher le nom et le prénom.

Dans les classes **Etudiant**, **EtudiantEtranger**, et **Professeur**, la méthode **afficher()** peut être définie avec le même nom et sera utilisé pour afficher les informations propres à chaque classe.

Pour ne pas répéter les instructions se trouvant dans la méthode de base, il faut utiliser le mot clé **super()**.

Redéfinition : Exemple 2

```
class Personne {
    private String nom, prenom;

    void afficher() {
        System.out.println("Nom :"+nom);
        System.out.println("Prenom :"+prenom);
    }
    ...
}

class Etudiant extends Personne {
    private String cne;
    void afficher() {
        super.afficher();
        System.out.println("CNE :"+cne);
    }
```

Redéfinition : Exemple 2 (suite)

```
class EtudiantEtranger extends Etudiant {
    private String nationalite;
    void afficher() {
        super.afficher();
        System.out.println("Nationalite :"+
            nationalite);
    }
    ...
}

class Professeur extends Personne{
    private String cin;
    void afficher() {
        super.afficher();
        System.out.println("CIN :"+cin);
    }
```

Remarques :

1. La méthode **super.afficher()** doit être la première instruction dans la méthode **afficher()**.
2. La méthode **super.afficher()** de la classe **EtudiantEtranger**, fait appel à **afficher()** de la classe **Etudiant**.
3. Si la classe **Etudiant** n'avait pas la méthode **afficher()**, alors, par transitivité, la méthode **super.afficher()** de la classe **EtudiantEtranger**, fait appel à **afficher()** de la classe **Personne**.
4. Il n'y a pas de : **super.super**.

Héritage et constructeurs

Une sous-classe n'hérite pas des constructeurs de la super-classe.

Exemple 1

Reprenons à la classe **Personne** et ajoutons à cette classe un seul constructeur.

Si aucun constructeur n'est défini dans les classes **Etudiant** et **Professeur**, il y aura erreur de compilation.

```
class Personne {
    private String nom, prenom;
    //Constructeur
    public Personne(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }
    ...
}
class Etudiant extends Personne {
    ...
    //Pas de constructeur
    ...
}
class Professeur extends Personne{
    ...
    //Pas de constructeur
    ...
```

Exemple 2

```
class Personne {
    private String nom, prenom;
    //Constructeur
    public Personne(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }
    ...
}
```

```
class Etudiant extends Personne {
    private String cne;
    //Constructeur
    public Etudiant(String nom, String prenom,
        String cne) {
         super(nom, prenom);
         this.cne = cne;
    }
    ...
}
class EtudiantEtranger extends Etudiant {
    private String nationalite;
    //Constructeur
    public EtudiantEtranger(String nom, String
        prenom, String cne, String nationalite) {
         super(nom, prenom,cne);
         this.nationalite = nationalite;
    }
```

Dans cette exemple, le constructeur de la classe **EtudiantEtranger** fait appel au constructeur de la classe **Etudiant** qui à son tour fait appel au constructeur de la classe **Personne**.

Exemple 3

```
class Rectangle{
    private double largeur;
    private double hauteur;
    public Rectangle(double l, double h) {
        largeur = l;
        hauteur = h;
    }
    ...
}

class Carre extends Rectangle {
    public Carre(double taille) {
        super(taille, taille);
    }
    ...
}
```

Remarques :

1. **super** doit être la première instruction dans le constructeur et ne doit pas être appelé 2 fois.
2. Il n'est pas nécessaire d'appeler **super** lorsque la super-classe admet un constructeur par défaut. Cette tâche sera réalisée par le compilateur.
3. Les arguments de super doivent être ceux d'un des constructeur de la super-classe.
4. Aucune autre méthode ne peut appeler **super(...)**.
5. Il n'y a pas de : **super.super**.

Opérateur « instanceof »

Types primitifs

Pour les types primitifs, les instructions suivantes sont vraies :

```
int i;
float x;
double y;
...
x = i;
y = x;
```

Un **int** est un **float** et un **float** est un **double** (conversion implicite).

Types primitifs

Par contre, les instructions suivantes sont fausses :

```
i = x;
x = y;
```

Un **float** n'est pas un **int** et un **double** n'est pas un **float**.

Pour les utiliser, il faut faire une conversion explicite (faire un **cast**) :

```
i = (int) x;
x = (float) y;
```

Objets

si **B** est sous-classe de **A**, alors on peut écrire :

```
//a est de type « A », mais l'objet référencé par a est de type « B ».
A a = new B(...);
A a1;
B b = new B();
a1 = b; // a1 de type « A », référence un objet de type « B »
```

Par contre, on ne peut pas avoir :

```
A a=new A();
B b;
b=a;
// erreur : on ne peut pas convertir du type « A » vers le type « B »
```

Cas des tableaux (Voir chapitre 7)

Reprenons l'exemple des classes **Personne**, **Etudiant**, **EtudiantEtranger** et **Professeur**. Les instructions suivantes sont vraies :

```
Etudiant e = new Etudiant();
EtudiantEtranger eEtr = new EtudiantEtranger();
Professeur prof = new Professeur();

Personne[] P = new Personne[3];
P[0] = e;
P[1] = eEtr;
P[2] = prof;

for (int i=0; i<3; i++)
    P[i].info();
```

Cas des tableaux

La méthode **info()** est ajoutée aux différentes classes pour indiquer dans quelle classe on se trouve :

```
void info(){
        System.out.println("Classe ...");
}
```

On reviendra plus en détails sur l'utilisation des tableaux dans le chapitre concernant les tableaux.

instanceof

Si « B » est une sous classe de « A » alors l'instruction :

b instanceof A ; retourne true.

```
Etudiant e = new Etudiant();
boolean b = e instanceof Personne; //b --> true

EtudiantEtranger eEtr =new EtudiantEtranger();
b = eEtr instanceof Personne;  //b --> true

Personne personne = new Etudiant();
b = personne  instanceof Etudiant; //b --> true

personne = new Personne();
b = personne  instanceof Etudiant; //b --> false
```

Cas de la classe **Object**

L'instruction :

personne instanceof Object;

retourne « true » car toutes les classes héritent, par défaut, de la classe Object

Chapitre 6
Polymorphisme et abstraction

Introduction

Le mot polymorphisme vient du grecque : **poly** (pour plusieurs) et **morph** (forme). Il veut dire qu'une même chose peut avoir différentes formes. Nous avons déjà vue cette notion avec la redéfinition des méthodes. Une même méthode peut avoir différentes définitions suivant la classe où elle se trouve.

Exemple introductif

Soient les classes **Personne**, **Etudiant** et **EtudiantEtranger** définies dans les chapitres précédents :

```
class Personne {
  ...
}
class Etudiant extends Personne {
    ...
}
class EtudiantEtranger extends Etudiant {
    private String nationalite;
    ...
}
```

Exemple introductif

Puisque un étudiant étranger est lui aussi un étudiant, au lieu de définir 2 tableaux :

```
Etudiant[] etudiants = new Etudiant[30];
EtudiantEtranger[] etudiantsEtrangers = new EtudiantEtranger[10];
```

on pourra définir un seul tableau comme suit :

```
Etudiant[] etudiants = new Etudiant[40];
```

Exemple introductif

Avant d'utiliser le tableau précédent, considérons la déclaration :

Personne personne;

Nous avons vu que les instructions suivantes sont toutes valides :

personne = new Personne();

personne = new Etudiant();

personne = new EtudiantEtranger();

De la même façon on peut initialiser le tableau de la façon suivante :

```
for(int int i=0; i<10; i++)
    etudiants[i] = new Etudiant();

etudiants[10] = new EtudiantEtranger();
etudiants[11] = new EtudiantEtranger();
...
```

Liaison dynamique

Considérons la classe **B** qui hérite de la classe **A** :

```
class A{
   public void message(){
      System.out.println("Je suis dans la classe A");
   }
}

class B extends A{
   public void message(){
     System.out.println("Je suis dans la classe B");
   }
   public void f(){
     System.out.println("Methode f de la classe B");
   }
}
```

Liaison dynamique

Considérons les instructions :

```
public static void main(String[] args) {
    A a = new A();
    B b = new B();
    a.message();
    b.message();
    b.f();
    a = new B();
    a.message();
}
```

Liaison dynamique

Dans l'exécution on aura le résultat suivant :

```
Je suis dans la classe A
Je suis dans la classe B
Methode f() de la classe B
Je suis dans la classe B
```

Lorsqu'une méthode est redéfinie (s'est spécialisée), c'est la version la plus spécialisée qui est appelée. La recherche de la méthode se fait dans la classe réelle de l'objet. La recherche s'est fait lors de l'exécution et non lors de la compilation. Ce processus s'appelle la **liaison dynamique**.

Il s'appelle aussi : **liaison tardive, dynamic binding, late-binding** ou **run-time binding**.

Remarques :

- Dans les instructions précédentes, la dernière instruction « a.message(); » fait appel à la méthode « message() » de la classe **B**.
- Si on ajoute l'instruction :
  a.f() ;
  après l'instruction :
  a = new B();
  on aura une erreur de compilation, du fait que la méthode « f() » n'est pas implémentée dans la classe de déclaration de l'objet « a » même si la classe réelle (la classe B) possède « f() ».
- Pour éviter l'erreur précédente, il faut faire un cast :
  ((B) a).f() ;

Remarques :

- La visibilité d'une méthode spécialisée peut être augmentée (par exemple de protected vers public) mais elle ne peut pas être réduite (par exemple de public vers private)

Méthodes de classes

Il n'y a pas de polymorphisme avec les méthodes de classes !

Exemple

```
class Etudiant {
    public void message(){
        System.out.println("Je suis un etudiant");
    }
}

class EtudiantEtranger extends Etudiant {
    public void message(){
        System.out.println("Je suis un etudiant
            etranger");
    }
}
```

Exemple

```
public class MethodesInstances {
    public static void main(String[] args){
        Etudiant e = new Etudiant();
        e.message();

        e = new EtudiantEtranger();
        e.message();
    }
}
```

L'exécution du programme précédent donnera :

```
Je suis un etudiant
Je suis un etudiant etranger
```

Si on modifie la méthode « **message()** » de la classe **Etudiant**, en la rendant statique :

```
class Etudiant {
    public static void message(){
        System.out.println("Je suis un etudiant");
    }
}

class EtudiantEtranger extends Etudiant {
    public static void message(){
        System.out.println("Je suis un etudiant
            etranger");
    }
}
```

```
public class MethodesInstances {
    public static void main(String[] args){
        Etudiant e = new Etudiant();
        e.message();
        e = new EtudiantEtranger();
        e.message();
    }
}
```

alors l'exécution du programme précédent donnera :

```
Je suis un etudiant
Je suis un etudiant
```

C'est la classe du type qui est utilisée (ici Etudiant) et non du type réel de l'objet (ici EtudiantEtranger).

Abstraction

Exemples introductifs

Reprenons les classes **Personne**, **Etudiant** et **EtudiantEtranger** et ajoutons aux classes **Etudiant** et **EtudiantEtranger** la méthode « **saluer()** » :

```
class Etudiant extends Personne {
    ...
    public void saluer() {
        System.out.println("Assalam alaikoum");
    }
}
class EtudiantEtranger extends Etudiant {
    private String nationalite;
    ...
    public void saluer() {
        System.out.println("Bonjour");
    }
}
```

Puisque la méthode « saluer() » est définie dans les deux classes **Etudiant** et **EtudiantEtranger**, on souhaite la définir dans la classe **Personne**.

Solution 1

Une première solution consiste à la définir comme suit :

```
class Personne {
    ...
    public void saluer() {
    }
}
```

Cette solution est mauvaise du fait que toute classe qui héritera de **Personne** pourra appeler cette méthode (qui ne fait rien !).

Solution 2

Une deuxième solution consiste à rendre la méthode « saluer() » abstraite dans **Personne** et par conséquent, obliger chaque classe qui hérite de **Personne** à définir sa propre méthode.

Remarques :

- Une méthode abstraite :
  - ne doit contenir que l'entête et doit être implémenté dans les sous classes ;
  - doit être public (ne peut pas être privée) ;
  - est déclarée comme suit :
    `public abstract typeRetour methAbstr(args);`
- Une méthode statique ne peut pas être abstraite.
- Une classe qui contient une méthode abstraite doit être elle aussi abstraite et doit être déclarée comme suit :
  `public abstract class nomClasse{ ... }`

Remarques :

- Une classe abstraite ne peut pas être utilisée pour instancier des objets. Une instruction telle que :
  obj=new nomClasse();
  est incorrecte (avec nomClasse est une classe abstraite).
- Une sous-classe d'une classe abstraite doit implémenter toutes les méthodes abstraites sinon elle doit être déclarée abstraite.

Solution 2 :

La classe **Personne** devient comme suit :

```
abstract class Personne {
    private String nom, prenom;
    public abstract void saluer();
    //...
}
```

Exemple 2

Considérons les classes **Cercle** et **Rectangle** qui sont des sous classes de la classe **FigureGeometrique**. Les surfaces d'un cercle et d'un rectangle ne sont pas calculées de la même façon.

→ Une solution consiste à définir dans la classe **FigureGeometrique** une méthode abstraite « surface() » et obliger les classes **Cercle** et **Rectangle** à implémenter cette méthode.

Exemple 2

```
abstract class FigureGeometrique {
    public abstract double surface();
}

class Cercle extends FigureGeometrique {
    private double rayon;

    public Cercle(double rayon) {
        this.rayon = rayon;
    }

    public double surface() {
        return Math.PI * rayon * rayon;
    }
}
```

Exemple 2

```
class Rectangle extends FigureGeometrique {
    public double largeur, longueur;

    public Rectangle(double large, double longue) {
        this.largeur = large;
        this.longueur = longue;
    }

    public double surface() {
        return largeur * longueur;
    }
}
```

Constructeurs et abstraction

Il est possible d'appeler une méthode abstraite dans le corps d'un constructeur, ceci est cependant déconseillé !

Exemple :

```
abstract class classeA {
    public abstract void m();

    public classeA() {
        m();
    }
}
```

Constructeurs et abstraction

```
class classeB extends classeA {
    private int b;
    public classeB() {
        // classeA() est invoquee implicitement
        // super(); (automatique)
        b = 1;
    }
    // definition de m pour classeB
    public void m() {
        System.out.println("b vaut : " + b);
    }
}
```

Constructeurs et abstraction

```
public class ConstructAbstraction {
    public static void main(String[] args) {
        classeB b = new classeB();
    }
}
```

Le résultat de l'exécution du programme précédent est :

```
b vaut : 0
```

Remarque

La classe **Math** n'est pas une classe abstraite même si on ne peut pas créer une instance de cette classe. Pour définir une classe non instanciable , il suffit de lui ajouter un et un seul constructeur **privé** sans arguments.

Remarque

Extrait de la classe **Math** :

```
public final class Math {

    /**
     * Don't let anyone instantiate this class.
     */
    private Math() {}

    ...
}
```

Toute classe abstraite est non instanciable mais l'inverse n'est pas vrai.

Chapitre 7
Tableaux

Déclaration et initialisation

Comme pour les autres langages de programmation, java permet l'utilisation des tableaux.

La déclaration d'un tableau à une dimension se fait de deux façons équivalentes :

`type tab[];` ou `type[] tab;`

`tab` est une référence à un tableau.

Exemple :

```
int tab [];
int [] tab;
```

Contrairement au langage C, la déclaration `int tab[10];` n'est pas permise. On ne peut pas fixer la taille lors de la déclaration.

Remarques :

En java, un tableau :

- est un objet ;
- est alloué dynamiquement (avec l'opérateur **new**) ;
- a un nombre fixe d'éléments de même type ;
- peut être vide (taille zéro).

Création

La création d'un tableau peut se faire soit, lors de la déclaration soit par utilisation de l'opérateur **new**.

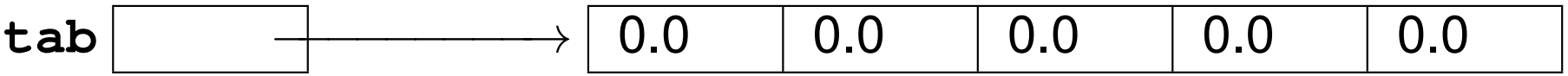
Exemples :

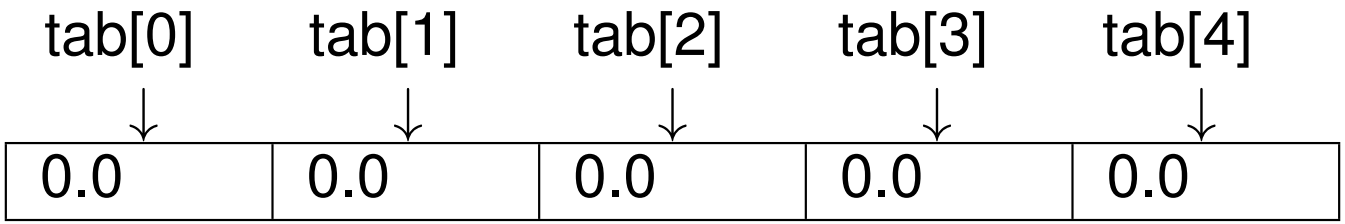
1. Création par initialisation au début :
   ```
   int [] tab={12,10,30*4};
   ```
2. Création par utilisation de **new** :
   ```
   int [] tab;
   tab = new int [5];
   ```
   ou bien :
   ```
   int [] tab = new int [5];
   ```

La déclaration

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

Crée un emplacement pour un tableau de 5 réels (double) et fait la référence à **`tab`** comme illustré par le schéma suivant :





Les valeurs du tableau sont initialisés aux valeurs par défaut (0 pour int, 0.0 pour double ...).

Déclaration mixte

On peut combiner la déclarations de tableaux avec d'autres variables de même type. La déclaration :

```
double scores[],  moyenne;
```

crée :

- un tableau non initialisé de type **double** nommé **scores** ;
- une variable de type **double** nommée **moyenne**.

Par contre, la déclaration :

```
double[]  scores,  moyenne;
```

crée deux tableaux non initialisés.

Taille d'un tableau

La taille d'un tableau est accessible par le champ « public final » length.

Exemple :

```
double[] scores = new double[10];
System.out.println(scores.length); // Affiche 10
```

Remarque :

La taille d'un tableau ne peut pas être changée. Par contre, la référence du tableau peut changer. Elle peut référencée un tableau de taille différente.

Exemple :

```
double[] tab1 = new double[10];
double[] tab2 = new double[5];
tab1 = new double[7];//tab1 est maintenant un nouveau tableau de
taille 7
tab2 = tab1;// tab1 et tab2 referencent le meme tableau
```

Parcours d'un tableau

Comme pour le langage C, on peut accéder aux éléments d'un tableau en utilisant l'indice entre crochets ([]).

Exemple

```
double [] tab = new double[10];
for(int i=0; i<tab.length;i++)
    carres[i]=i*i;

for(int i=0; i<tab.length;i++)
    System.out.printf("tab[%d] = %.2f\n",i,tab[i]);
```

Remarque :

La lecture d'un tableau peut se faire de la façon suivante :

```
for(double carre: tab)
    System.out.println(carre);
```

Cette méthode de parcours d'un tableau n'est valable que pour la lecture et ne peut pas être utilisée pour la modification. Elle a l'avantage d'éviter l'utilisation des indices.

Correspond à : « pour chaque élément ... » (for each ...)

Exemple

Dans le listing suivant, pour calculer la somme des éléments du tableau, on n'a pas besoin de connaître les indices.

```
double somme = 0.0;
for(double carre : tab)
    somme += carre;

System.out.println("Somme ="+somme);
```

Parcours des arguments de la ligne de commande

La méthode principale « main() » contient un tableau de « String ». Ce tableau peut être utilisé comme les autres tableaux. Pour afficher son contenu, on peut utiliser le code suivant :

```
System.out.println("taille de args "+args.length);
for(String arg :  args)
    System.out.print(arg.toUpperCase()+" ");
```

Si le programme s'appel « Test » et on exécute la commande :

```
java Test bonjour tous le monde
```

alors le code précédent affichera :

```
taille de args : 4
BONJOUR TOUS LE MONDE
```

Copie de tableaux

Comme on l'a vu précédemment, pour copier le contenu d'un tableau « tab2 » dans autre tableau « tab1 », l'instruction :

tab1 = tab2;

ne copie que la référence et non le contenu. Pour copier le contenu, on peut soit, procéder de façon classique et copier élément par élément, soit utiliser des méthodes prédéfinies.

Utilisation de : System.arraycopy()

Sa syntaxe est :

System.arraycopy(src, srcPos, dest, destPos, nb);

Les arguments sont définis comme suit :

- **src** : tableau source ;
- **srcPos** : indice du premier élément copié à partir de src ;
- **dest** : tableau destination ;
- **destPos** : indice de destination où sera copié le premier élément ;
- **nb** : nombre d'éléments à copier.

Exemple

```
double [] carres = new double[10];
double [] carresBis = new double[10];
double [] carresTest = new double[6];

for(int i=0; i<carres.length;i++)
    carres[i]=i*i;

//Copie de carres dans carresBis
System.arraycopy(carres, 0, carresBis, 0, 10);

//Copie de 6 elements de carres a partir de l'indice
    4 dans carresTest
//(a partir du premier element)
System.arraycopy(carres, 4, carresTest, 0, 6);
```

Utilisation de : Arrays.copyOf() ou Arrays.copyOfRange()

La classe **java.util.Arrays** contient différentes méthodes pour la manipulation des tableaux (copie, trie, ...). Pour copier un tableau :

- en entier, on peut utiliser la méthode **Arrays.copyOf()** ;
- juste une partie, on peut utiliser la méthode **Arrays.copyOfRange()**.

Exemple

```
import java.util.Arrays;
...
double [] t1 = new double[10];

for(int i = 0; i < t1.length;i++)
    t[i] = i*i;

int[] t2 = Arrays.copyOf(t1, 10);
```

1. crée le tableau t2
2. affecte au tableau t2 les 10 premiers éléments du tableau tab.

Arrays.copyOfRange

```
int[] t3= Arrays.copyOfRange(t1, debut, fin);
```

1. crée le tableau t3
2. affecte à t3 les éléments de t1 situés entre les indices : debut et (fin-1) :

Comparer le contenu de deux tableaux de types primitifs

Comme pour le cas du copie, pour comparer le contenu de deux tableaux de types primitifs, on peut, soit :

- procéder de façon classique et comparer les éléments des deux tableaux un à un ;
- utiliser la méthode **equals()** de la classe **Arrays**.

Exemple :

Soient t1 et t2 deux tableaux de primitifs déjà initialisés :

```
import java.util.Arrays;
...
boolean b=Arrays.equals(tab1,tab2);
```

Utilisation de la classe Arrays

La classe **Arrays** contient différentes méthodes pour la manipulation des tableaux. Nous avons déjà utilisé les méthodes **Arrays.copyOf()**, **Arrays.copyOfRange()**, et **Arrays.equals()**. Dans ce qui suit, nous allons voir quelques méthodes pratiques :

Trie

La méthode **void sort(type[] tab)**, permet de trier le tableau **tab** par ordre croissant. Le résultat du trie est retourné dans **tab**.

La méthode **void sort(type[] tab, int indiceDeb, int indiceFin)** permet de trier le tableau **tab** par ordre croissant à partir de l'indice **indiceDeb** (inclue) jusqu'au **indiceFin** (exclue)

Exemple

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

for(int i=0; i<10;i++)
    tab[i] = Math.random()*10;
//Math.random() : genere un nombre aleatoire.

Arrays.sort(tab);

for(int i=0; i<10;i++)
    tab[i] = Math.random()*10;

Arrays.sort(tab, 2, 5);
//Trie les elements tab[2], tab[3] et tab[4] par
   ordre croissant
```

Recherche

La méthode **int binarySearch(type [] a, type val)**, permet de chercher **val** dans le tableau **tab**. Le tableau doit être trié par ordre croissant, sinon, le résultat de retour sera indéterminé.

Le résultat de retour est :

- l'indice du tableau qui contient **val** si le tableau contient **val** ;
- une valeur négative si le tableau ne contient pas **val**.

La méthode **int binarySearch(type [] a, int indiceDeb, int indiceFin, type val)**, permet de chercher **val** dans l'intervalle du tableau **tab** entre l'indice **indiceDeb** (inclue) et **indiceFin** (exclue).

Exemple

```
double[] tab = new double[10];
double val=5;

for(int i=0; i<10;i++)
    tab[i] = Math.random()*10;

tab[3] = val;

Arrays.sort(tab);

//Recherche dans le tableau
System.out.println(Arrays.binarySearch(tab, val));

//Recherche dans l'intervalle 2..7
System.out.println(Arrays.binarySearch(tab, 2, 7,
   val));
```

Remplissage

La méthode **void fill(type[] tab, type val)** affecte **val** à tous les éléments du tableau.

La méthode **void fill(type[] tab, int indiceDeb, int indiceFin, type val)** affecte **val** aux éléments du tableau compris entre **indiceDeb** et **indiceFin-1**.

```
double[] tab = new double[10];
double val=10;

Arrays.fill(tab, val);

val=7;
//tab[2]=tab[3]=tab[4]=7
Arrays.fill(tab,2,5,val);
```

Méthode toString()

La méthode **void toString(type[] tab)** permet de convertir le tableau en une chaîne de caractères. Elle met le tableau entre [ ] avec les valeurs séparés par « , » (virgule suivie par espace).

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

System.out.println(Arrays.toString(tab));
//Affiche : [1, 2, 3, 4]
```

Passage d'un tableau dans une méthode

Les tableaux sont des objets et par conséquent lorsqu'on les passe comme arguments à une méthode, cette dernière obtient une copie de la référence du tableau et par suite elle a accès aux éléments du tableau. Donc, toute modification affectera le tableau.

Exemple

Le programme suivant :

```
import java.util.Arrays;

public class TabMethodesArg {
    public static void main(String[] args) {
        int[] tab = { 1, 2, 3, 4 };

        System.out.print("Debut de main: ");
        System.out.println(Arrays.toString(tab));

        // Appel de test
        test(tab);

        System.out.print("Fin de main: ");
        System.out.println(Arrays.toString(tab));
    }
```

Exemple (suite)

```
    public static void test(int[] x) {
        System.out.println("Debut :\t" + Arrays.
            toString(x));
        Arrays.fill(x, 10);
        System.out.println("fin :\t" + Arrays.
            toString(x));
    }
}
```

affichera :

```
Debut de main: [1, 2, 3, 4]
Debut :    [1, 2, 3, 4]
fin :    [10, 10, 10, 10]
Fin de main: [10, 10, 10, 10]
```

Retour d'un tableau dans une méthode

Une méthode peut retourner un tableau.

Exemple

La méthode :

```
public static int [] scoreInitial() {
    int score[] ={2, 3, 6, 7, 8};
    return score;
}
```

peut être utilisé comme suit :

```
public static void main(String[] args) {
    int[] tab = scoreInitial();

    System.out.println(Arrays.toString(tab));
}
```

Tableaux d'objets

L'utilisation des tableaux n'est pas limité aux types primitifs. On peut créer des tableaux d'objets. On a déjà utilisé les tableaux d'objets dans la méthode « main(String[] args) », puisque « String » est une classe.

Considérons la classe « Etudiant » vue dans les chapitres précédents et en TP :

```
class Etudiant {
    private String nom, prenom, cne;
    public Etudiant() {
    }
    ...
}
```

Tableaux d'objets

La déclaration :

Etudiant[] etudiants = new Etudiant[30];

Crée l'emplacement pour contenir 30 objets de type « Etudiant ». Elle ne crée que les références vers les objets. Pour créer les objets eux mêmes, il faut utiliser, par exemple, l'instruction suivante :

```
for(int i=0; i<etudiants.length; i++)
    etudiants[i]=new Etudiant();
```

Attention

Pour les tableaux d'objets, les méthodes de la classe « **Arrays** » opèrent sur les références et non sur les valeurs des objets.

Exemple

```
import java.util.Arrays;

public class TableauxObjets {
    public static void main(String[] args) {
        Etudiant [] etud1 = new Etudiant[2];
        Etudiant [] etud2 = new Etudiant[2];
        Etudiant [] etud3 = new Etudiant[2];
        boolean b;
        //Initialisation de etud1
        etud1[0]= new Etudiant("Oujdi","Ali");
        etud1[1]= new Etudiant("Berkani","Lina");

        //Initialisation de etud2
        etud2[0]= new Etudiant("Mohammed","Ali");
        etud2[1]= new Etudiant("Figuigui","Fatima");
```

Exemple (suite)

```
        b = Arrays.equals(etud1, etud2);
        System.out.println(b); //affiche false

        etud2[0] = etud1[0];
        etud2[1] = etud1[1];
        b = Arrays.equals(etud1, etud2);
        System.out.println(b); //affiche true

        etud3 = etud1;
        b = Arrays.equals(etud1, etud3);
        System.out.println(b); //affiche true
    }
}
```

Objets qui contiennent des tableaux

On peut avoir un objet qui contient des tableaux.

Considérons la classe « Etudiant » et ajoutons à cette classe le tableau **notes** comme suit :

```
class Etudiant {
    private String nom, prenom, cne;
    private double [] notes = new double[6];
    ...
}
```

Tableaux à plusieurs dimensions

On peut créer des tableaux à plusieurs dimensions par ajout de crochets (`[ ]`). Par exemple, l'instruction :

```
double [][] matrice;
```

déclare un tableau à 2 dimensions de type **double**.

Comme pour le tableau à une seule dimension, la création d'un tableau multi-dimensionnelle peut se faire par utilisation de l'opérateur **new**.

Exemple

L'instruction :

matrice = new double[4][3];

crée un tableau de 4 lignes et 3 colonnes.

On peut combiner les deux instructions précédentes :

double [][] matrice = new double[4][3];

Remarques

- En langage C, un tableau à plusieurs dimensions est en réalité un tableau à une dimension. Par exemple, la déclaration :
  double matrice [4][3];
  crée en mémoire un tableau (contiguë) de 12 double.
- En Java, un tableau à plusieurs dimensions n'est pas contiguë en mémoire. En Java, un tableau de plusieurs dimensions est un tableau de tableaux.
- On peut définir un tableau à 2 dimensions dont les colonnes n'ont pas la même dimension.

Exemple 1

```
double[][] tabMulti = new double[2][];

//tabMulti[0] est un tableau de 3 doubles
tabMulti[0] = new double[3];

//tabMulti[1] est un tableau de 4 doubles
tabMulti[1] = new double[4];

System.out.println(tabMulti.length);     //Affiche 2
System.out.println(tabMulti[0].length); //Affiche 3
System.out.println(tabMulti[1].length); //Affiche 4
```

Exemple 1 (suite)

```
for(int i=0;i<tabMulti.length;i++)
    for(int j=0; j<tabMulti[i].length;j++)
        tabMulti[i][j]=i+j;


System.out.println(Arrays.deepToString(tabMulti));
//Affiche [[0.0, 1.0, 2.0], [1.0, 2.0, 3.0, 4.0]]
```

Exemple 2

Dans l'exemple suivant, on va créer un tableau triangulaire qui sera initialisé comme suit :

```
0
0   0
0   0   0
0   0   0   0
```

Exemple 2

```
final int N = 4;
int [][] tabTriangulaire = new int[N][];
for(int n=0; n<N; n++)
    tabTriangulaire[n]= new int[n+1];

System.out.println(Arrays.deepToString(
   tabTriangulaire));
//Affiche [[0], [0, 0], [0, 0, 0], [0, 0, 0, 0]]

for(int i=0;i<tabTriangulaire.length;i++){
    for(int j=0; j<tabTriangulaire[i].length;j++)
        System.out.print(tabTriangulaire[i][j]+"\t")
            ;
    System.out.println();
}
```

Parcours d'un tableau multi-dimensionnel

Comme pour le cas à une seule dimension, on peut utiliser la boucle (pour chaque -for each) pour accéder au contenu d'un tableau multi-dimensionnel.

```
double [][] matrice = new double[4][3];

for(int i=0; i<4; i++)
    for(int j=0; j<3;j++)
            matrice[i][j]=i+j;

double somme=0;
for(double [] ligne : matrice)
    for(double val: ligne)
        somme += val;

System.out.println("somme = "+somme);
```

Exemple

```
double[][] tabMulti = {new double[4],new double[5]};

System.out.println(tabMulti.length); //Affiche 2
System.out.println(tabMulti[0].length); //Affiche 4
System.out.println(tabMulti[1].length); //Affiche 5


double[][] tabMulBis = {{1,2},{3,5},{3,7,8,9,10}};

System.out.println(tabMulBis.length); //Affiche 3
System.out.println(tabMulBis[0].length); //Affiche 2
System.out.println(tabMulBis[1].length); //Affiche 2
System.out.println(tabMulBis[2].length); //Affiche 5
```

Chapitre 8
Chaînes de caractères

Introduction

Considérons l'exemple suivant :

```
import java.util.Scanner;

public class TestComparaisonChaines {
  public static void main(String[] args) {
    String nom = "smi", str;
    Scanner input = new Scanner(System.in);
    System.out.print("Saisir le nom : ");
    str = input.nextLine();
    if (nom == str)
      System.out.println(nom + " = " + str);
    else
      System.out.println(nom + " different de " +
          str);
  }
}
```

Introduction

Dans cet exemple, on veut comparer les deux chaînes de caractères `nom` initialisée avec `"smi"` et `str` qui est saisie par l'utilisateur.

Lorsque l'utilisateur exécute le programme précédent, il aura le résultat suivant :

```
Saisir le nom de la filiere : smi
smi different de smi
```

Il semble que le programme précédent produit des résultats incorrects. Le problème provient du fait, que dans java, **String** est une classe et par conséquent chaque chaîne de caractères est un objet.

Remarque :

L'utilisation de l'opérateur == implique la comparaison entre les références et non du contenu.

En java, il existe des classes qui permettent la manipulation des caractères et des chaînes de caractères :

- **Character :** une classe qui permet la manipulation des caractères (un seul caractère).
- **String :** manipule les chaînes de caractères fixes.
- **StringBuilder** et **StringBuffer** : manipulent les chaînes de caractères modifiables.

Manipulation des caractères

Nous donnons quelques méthodes de manipulation des caractères.

L'argument passé pour les différentes méthodes peut être un caractère ou son code unicode.

Majuscule

isUpperCase() : test si le caractère est majuscule

toUpperCase() : si le caractère passé en argument est une lettre minuscule, elle retourne son équivalent en majuscule. Sinon, elle retourne le caractère sans changement.

Majuscule : Exemple

```
public class TestUpper {
    public static void main(String[] args) {
        char test='a';

        if (Character.isUpperCase(test))
            System.out.println(test + " est
                majuscule");
        else
            System.out.println(test + " n'est pas
                majuscule");

        test = Character.toUpperCase(test);
        System.out.println("Apres toUpperCase() : "
            + test);
    }
}
```

Minuscule

isLowerCase() : test si le caractère est minuscule

toLowerCase() : Si le caractère passé en argument est une lettre majuscule, elle retourne son équivalent en minuscule. Sinon, elle retourne le caractère sans changement.

isDigit() : Retourne `true` si l'argument est un nombre (0−9) et `false` sinon

isLetter() : Retourne `true` si l'argument est une lettre et `false` sinon

isLetterOrDigit() : Retourne `true` si l'argument est un nombre ou une lettre et `false` sinon

isWhitespace() : Retourne `true` si l'argument est un caractère d'espacement et `false` sinon. Ceci inclue l'espace, la tabulation et le retour à la ligne

Déclaration

Comme on l'a vu dans les chapitres précédents, la déclaration d'une chaîne de caractères se fait comme suit :

```
String  nom;
```

L'initialisation se fait comme suit :

```
nom="Oujdi";
```

Les deux instructions peuvent être combinées :

```
String  nom = "Oujdi";
```

L'opérateur **new** peut être utilisé :

```
String  nom = new String("Oujdi");
```

Pour créer une chaîne vide : `String  nom = new String();`

ou bien : `String  nom = "";`

Remarques

Une chaîne de type "Oujdi" est considérée par java comme un objet.
Les déclarations suivantes :

```
String  nom1 = "Oujdi";
String  nom2 = "Oujdi";
```

déclarent deux variables qui référencent le même objet ("Oujdi").

Par contre, les déclarations suivantes :

```
String  nom1 = new String("Oujdi");
String  nom2 = new String("Oujdi"); // ou nom2 = new String(nom1)
```

déclarent deux variables qui référencent deux objets différents.

Méthodes de traitement des chaînes de caractères

La compilation du programme suivant génère l'erreur « `array required, but String found System.out.println("Oujdi"[i]);` »

```
public class ProblemeManipString {
    public static void main(String[] args) {
        for(int i=0; i<5; i++)
            System.out.println("Oujdi"[i]);
    }
}
```

Pour éviter les erreurs de ce type, la classe « String » contient des méthodes pour manipuler les chaînes de caractères. Dans ce qui suit, nous donnons quelques unes de ces méthodes.

Méthode **charAt()**

Retourne un caractère de la chaîne.

Une correction de l'exemple précédent est :

```
public class ProblemeManipStringCorrection {
    public static void main(String[] args) {
        for(int i=0; i<5; i++)
            System.out.println("Oujdi".charAt(i));
    }
}
```

Méthode **concat()**

Permet de concaténer une chaîne avec une autre.

```
nom = "Oujdi".concat(" Mohammed");
//nom<--"Oujdi Mohammed"
```

Méthode **trim()**

supprime les séparateurs de début et de fin (espace, tabulation, ...)

```
nom = "\n Oujdi"+" Mohammed \n\t";
nom = nom.trim(); //nom<--"Oujdi Mohammed"
```

Méthodes **replace()** et **replaceAll()**

- **replace()** : Remplace toutes les occurrences d'une chaîne de caractères avec une autre chaîne de caractères
- **replaceAll()** : Remplace toutes les occurrences d'une expression régulière par une chaîne

```
String str;
str = "Bonjour".replace( "jour", "soir" );
//str <--"Bonsoir"
str = "soir".replaceAll("[so]", "t");
//str <-- "ttir"
str = "def".replaceAll("[a-z]","A");
//str <-- "AAA"
```

Méthode compareTo()

Permet de comparer un chaîne avec une autre chaîne.

```
String abc = "abc";
String def = "def";
String num = "123";
if ( abc.compareTo( def ) < 0 )    // true
  if ( abc.compareTo( abc ) == 0 )  // true
    if ( abc.compareTo( num ) > 0 )    // true
      System.out.println(abc);
```

Méthode **indexOf()**

Cherche la première occurrence d'un caractère ou d'une sous-chaîne de la chaîne

```
String abcs = "abcdefghijklmnopqrstuvwxyz";
int i = abcs.indexOf( 's' );   // 18
int j = abcs.indexOf( "def" ); // 3
int k = abcs.indexOf( "smi" );   // -1
```

Méthode **valueOf()**

Retourne la chaîne qui est une représentation d'une valeur

```
double x=10.2;
str = String.valueOf(x); //str <-- "10.2"
int i = 20;
str = String.valueOf(i); //str <-- "20"
```

Méthode **substring()**

Retourne une sous-chaîne de la chaîne :

- **substring(debut) :** la sous-chaîne commence à partir du caractère qui se trouve à l'indice **debut** et se termine à la fin de la chaîne ;
- **substring(debut, fin) :** la sous-chaîne commence à partir du caractère qui se trouve à l'indice **debut** et se termine au caractère qui se trouve à l'indice **fin-1**.

Exemple :

```
str = "Bonjour".substring(3); //str <-- "jour"
str = "toujours".substring(3,7); //str <-- "jour"
```

Méthodes **startsWith()** et **endsWith()**

- **startsWith()** : Vérifie si une chaîne commence par un suffixe
- **endsWith()** : Vérifie si une chaîne se termine par un suffixe

```
String url = "http://www.ump.ma";
if ( url.startsWith("http") ) // true
  if ( url.endsWith("ma") ) // true
      System.out.println(str);
```

Méthodes equals() et equalsIgnoreCase()

La méthode :

- **equals()** compare une chaîne avec une autre chaîne en tenant compte de la casse (majuscule ou minuscule) ;
- **equalsIgnoreCase()** compare une chaîne avec une autre chaîne en ignorant la casse.

Exemple :

```
String str1="test", str2="Test";
if(str1.equals(str2))
   System.out.println("Les 2 chaines sont egaux");
else
   System.out.println("Les 2 chaines differents");

if(str1.equalsIgnoreCase(str2))
   System.out.println("Les 2 chaines sont egaux (
      sans tenir compte de la casse)");
```

Méthode getBytes()

Copie les caractères d'une chaîne dans un tableau de bytes.

Exemple :

```
String str1="abc_test_123";
byte[] b;
b = str1.getBytes();
for (int i = 0; i < b.length; i++)
  System.out.println(b[i]);
```

Méthode getChars()

Copie les caractères d'une chaîne dans un tableau de caractères.

Utilisation :
Soit **str** une chaîne de caractères. La méthode **getChars()** s'utilise comme suit :

```
str .getChars(srcDebut, srcFin, dst,  dstDebut)
```

Paramètres :

- **srcDebut :** indice du premier caractère à copier ;
- **srcFin :** indice après le dernier caractère à copier ;
- **dst :** tableau de destination ;
- **dstDebut :** indice du premier élément du tableau de destination.

Méthode getChars() :
Exemple :

```
str = "Ceci est un test";
char [] c = new char[str.length()];

str.getChars(0, str.length(), c, 0);
for (int i = 0; i < c.length; i++)
    System.out.println(c[i]);
```

Méthode toCharArray()

Met la chaîne dans un tableau de caractères.

Exemple :

```
str="Test";
char[] tabC=str.toCharArray();
for (int ind = 0; ind < tabC.length; ind++)
    System.out.println(tabC[ind]);
```

Méthode isEmpty()

Retourne **true** si la chaîne est de taille nulle.

Exemple :

```
if(str.isEmpty())
    System.out.println("Chaine vide");
else
    System.out.println("Chaine non vide");
```

Méthode indexOf()

Cherche la première occurrence d'un caractère ou d'une sous-chaîne de la chaîne. Elle retourne l'indice du premier occurrence du caractère dans la chaîne ou **-1** si le caractère n'existe pas.

Exemple :

```
String abcs = "abcdefghijklmnopqrstuvwxyz";
int i = abcs.indexOf( 's' );   // 18
int j = abcs.indexOf( "def" ); // 3
int k = abcs.indexOf( "smi" );   // -1
```

Méthode lastIndexOf()

Cherche la dernière occurrence d'un caractère ou d'une sous-chaîne dans une chaîne. Elle retourne l'indice du dernier occurrence du caractère dans la chaîne ou **-1** si le caractère n'existe pas.

Exemple :

```
String abcs = "abcdefghijklmnopqrstuvwxyz+str";
int i = abcs.lastIndexOf( 's' );   // 27
int j = abcs.lastIndexOf( "def" ); // 3
int k = abcs.lastIndexOf( "smi" );   // -1
```

Méthode replaceFirst()

Remplace la première occurrence d'une expression régulière par une chaîne.

Exemple :

```
str = "Test1test2".replaceFirst("[0-9]", "_");
//Test_test2
```

Méthodes toLowerCase() et toUpperCase()

La méthode :

- **toLowerCase()** convertie la chaîne en minuscule ;
- **toUpperCase()** convertie la chaîne en majuscule.

Exemple :

```
String nom = "Oujdi";
nom = nom.toUpperCase(); //nom <-- "OUJDI"

str = "TEst";
str = str.toLowerCase(); //str <-- "test"
```

Méthode split()

split() sépare la chaîne en un tableau de chaînes en utilisant une expression régulière comme délimiteur.

```
nom = "Oujdi Mohammed";
String[] tabStr = nom.split(" ");
for (int ind = 0; ind < tabStr.length; ind++)
  System.out.println(tabStr[ind]); //Affichera :
//Oujdi
//Mohammed
str="Un:deux,trois;quatre";
tabStr = str.split("[:,;]");
for (int ind = 0; ind < tabStr.length; ind++)
    System.out.println(tabStr[ind]);  //Affichera :
//Un
//deux
//trois
//quatre
```

Méthode hashCode()

Retourne le code de hachage (hashcode) d'une chaîne. Pour une chaîne **s**, le code est calculé de la façon suivante :

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \cdots + s[n-1]$$

avec : $s[i]$ est le code du caractère à la position i (voir la méthode **getBytes()**).

Exemple :

```
str = "12";
System.out.println("code " + str.hashCode());
//retourne : 1569 = 49*31+50
```

Méthode valueOf()

Retourne la chaîne qui est une représentation d'une valeur.

Exemple :

```
double x = 10.2;
str = String.valueOf(x); //str <-- "10.2"

int test = 20;
str = String.valueOf(test); //str <-- "20"
```

Conversion entre String et types primitifs

Comme on l'a vu précédemment, la méthode **`valueOf()`** de la classe **`String`** permet de récupérer la valeur d'un type primitif.

Les classes **Byte, Short, Integer, Long, Float,** et **Double**, disposent de la méthode statique **toString()** qui permet de convertir un type primitif vers un **String**. Elles disposent respectivement des méthodes statiques **parseByte(), parseShort(), parseInt(), parseLong(), parseFloat()** et **parseDouble()**, qui permettent de convertir une chaîne en un type primitif.

Exemple :

```
public class TestConversion{
  public static void main(String[] args){
    double x = 4.5;
    String str = Double.toString(x);//str<--"4.5"

    int test = 20;
    str = Integer.toHexString(test); //str <-- "20"

    str = "2.3";
    x = Double.parseDouble(str); //x <-- 2.3

    str = "5";
    test = Integer.parseInt(str); //test <-- 5
  }
}
```

Affichage des objets

Le code suivant :

```
Etudiant etud = new Etudiant("Oujdi","Ali","A20");
System.out.println(etud);
```

affichera quelque chose comme : `Etudiant@1b7c680`.

Cette valeur correspond à la référence de l'objet.

Affichage des objets

Si on souhaite afficher le contenu de l'objet en utilisant le même code, il faut utiliser la méthode **toString** prévu par Java.

Affichage des objets : Exemple

```
public class TestEtudiantToString {
  public static void main(String[] args) {
    Etudiant et = new Etudiant("Oujdi","Ali","A20");
    System.out.println(etud);
  }
}

class Etudiant {
  private String nom, prenom, cne;
  ...
  public String toString(){
    return "Nom : "+nom+"\nPrenom : "+prenom+"\nCNE
        : "+cne;
  }
  ...
}
```

Affichage des objets : Exemple

Affichera :

```
Nom : Oujdi
Prenom : Ali
CNE : A20
```

Comparaison des objets

Le code suivant :

```
Etudiant etud1 = new Etudiant("Oujdi","Ali","A20");
Etudiant etud2 = new Etudiant("Oujdi","Ali","A20");

if(etud1 == etud2)
    System.out.println("Identiques");
else
    System.out.println("Differents");
```

affichera toujours « `Differents` », du fait que la comparaison s'est faite entre les références des objets.

Comparaison des objets

Comme pour la méthode **toString**, Java prévoit l'utilisation de la méthode **equals** qui pourra être définie comme suit :

```
class Etudiant {
  private String nom, prenom, cne;
  ...
  public boolean equals(Etudiant bis){
    if (nom == bis.nom && prenom == bis.prenom  &&
        cne == bis.cne)
        return true;
    else
        return false;
  }
}
```

Comparaison des objets : utilisation de equals

```
public class TestEtudiantToString {
  public static void main(String[] args) {
  Etudiant et = new Etudiant("Oujdi","Ali","A20");
  Etudiant et1 = new Etudiant("Oujdi","Ali","A20");

  if(et.equals(et1))
    System.out.println("Identiques");
  else
    System.out.println("Differents");
  }
}
```

Les classes StringBuilder et StringBuffer

Lorsqu'on a dans un programme l'instruction

```
str = "Bonjour";
```

suivie de

```
str = "Bonsoir";
```

le système garde en mémoire la chaîne "Bonjour" et crée une nouvelle place mémoire pour la chaîne "Bonsoir". Si on veut modifier "Bonsoir" par "Bonsoir SMI5", alors l'espace et "SMI5" ne sont pas ajoutés à "Bonsoir" mais il y aura création d'une nouvelle chaîne. Si on fait plusieurs opérations sur la chaîne str, on finira par créer plusieurs objets dans le système, ce qui entraîne la consommation de la mémoire inutilement.

Les classes StringBuilder et StringBuffer

Pour remédier à ce problème, on peut utiliser les classes **StringBuilder** ou **StringBuffer**. Les deux classes sont identiques à l'exception de :

- **StringBuilder** : est plus efficace.
- **StringBuffer** : est meilleur lorsque le programme utilise les threads.

Puisque tous les programmes qu'on va voir n'utilisent pas les threads, le reste de la section sera consacré à la classe **StringBuilder**.

Déclaration et création

Pour créer une chaîne qui contient "Bonjour", on utilisera l'instruction :

```
StringBuilder  message = new StringBuilder("Bonjour");
```

Pour créer une chaîne vide, on utilisera l'instruction :

```
StringBuilder  message = new StringBuilder();
```

Remarque :

L'instruction :

```
StringBuilder  message = "Bonjour";
```

est incorrecte. Idem, si « message » est un StringBuilder, alors l'instruction :

```
message = "Bonjour";
```

est elle aussi incorrecte.

Méthodes de StringBuilder

length()	Retourne la taille de la chaîne
charAt()	Retourne un caractère de la chaîne
substring()	Retourne une sous-chaîne de la chaîne
setCharAt(i,c)	permet de remplacer le caractère de rang i par le caractère c.
insert(i,ch)	permet d'insérer la chaîne de caractères ch à partir du rang i
append(ch)	permet de rajouter la chaine de caractères ch à la fin
deleteCharAt(i)	efface le caractère de rang i.
toString()	Convertie la valeur de l'objet en une chaîne (conversion de StringBuilder vers String)
concat()	Concaténer une chaîne avec une autre
contains()	Vérifie si une chaîne contient une autre chaîne

endsWith()	Vérifie si une chaîne se termine par un suffixe
equals()	Compare une chaîne avec une autre chaîne
getBytes()	Copie les caractères d'une chaîne dans un tableau de bytes
getChars()	Copie les caractères d'une chaîne dans un tableau de caractères
hashCode()	Retourne le code de hachage (hashcode) d'une chaîne
indexOf()	Cherche la première occurrence d'un caractère ou d'une sous-chaîne de la chaîne
lastIndexOf()	Cherche la dernière occurrence d'un caractère ou d'une sous-chaîne dans une chaîne
replace()	Remplace toutes les occurrences d'un caractère avec un autre caractère

Remarque :

On ne peut pas faire la concaténation avec l'opérateur + entre des StringBuilder. Par contre :

StringBuilder + String

produit une nouvelle chaîne de type String.

Exemple

```
public class TestStringBuilder {
  public static void main(String args[]) {
  StringBuilder strBuilder = new StringBuilder("
     Bonjour SMI5");
  int n = strBuilder.length(); // n <-- 12

  char c = strBuilder.charAt(2); // c <-- 'n'

  strBuilder.setCharAt(10, 'A');
  // remplace dans strBuilder le caractere 'I' par '
     A'.
  // strBuilder <-- "Bonjour SMA5"
```

Exemple (suite)

```
strBuilder.insert(10, " semestre ");
// insere dans strBuilder la chaine " semestre "
   a
// partir du rang 10.
// strBuilder <-- "Bonjour SMA semestre 5"


strBuilder.append(" (promo 14-15)");
// strBuilder <-- "Bonjour SM semestre A5 (promo
   14-15)"

strBuilder = new StringBuilder("Boonjour");
strBuilder.deleteCharAt(2);
// supprime de la chaine strBuilder le caractere
   de rang 2.
// strBuilder <-- "Bonjour"
```

Exemple (suite)

```
    String str = strBuilder.toString();
    // str <-- "Bonjour"

    str = strBuilder.substring(1, 4);
    // str <-- "onj"

    str = strBuilder + " tous le monde";
    // str <-- "Bonjour tous le monde"
    }
}
```

Exercices

Exercice

Écrivez une application qui compte le nombre d'espaces contenus dans une chaîne de caractères saisie par l'utilisateur. Sauvegardez le fichier sous le nom **NombreEspaces.java**.

Solution

```
import java.util.Scanner;

public class NombreEspaces {
  public static void main(String[] args) {
    int nombreEspaces = 0;
    String str;

    Scanner clavier = new Scanner(System.in);
    System.out.println("Saisir une chaine ");
    str = clavier.nextLine();

    for (int i = 0; i < str.length(); i++)
      if (str.charAt(i) == ' ')
        nombreEspaces++;
    System.out.println("Nombre d'espaces : " +
       nombreEspaces);
```

Nombre de caractères d'espacement :

```
        nombreEspaces = 0;
        for (int i = 0; i < str.length(); i++)
          if (Character.isWhitespace(str.charAt(i)))
            nombreEspaces++;

        System.out.println("Nombre de caracteres d'
            espacement : " + nombreEspaces);
        clavier.close();
    }
}
```

Exercice

L'utilisation de trois lettres dans les acronymes est courante. Par exemple :

- JDK : Java Development Kit ;
- JVM :Java Virtual Machine ;
- RAM : Random Access Memory ;
- ...

Écrivez un programme qui demande à l'utilisateur de saisir trois mots et affiche à à l'écran l'acronyme correspondant composé des trois premières lettres en majuscule. Si l'utilisateur saisisse plus de trois mots, le reste sera ignoré.

Solution

```
import java.util.Scanner;

public class Acronymes {
    public static void main(String[] args) {
        String str, acronyme = "";
        String[] tabStr;

        Scanner clavier = new Scanner(System.in);
        do {// pour forcer l'utilisateur a saisir plus
            de 3 mots
            System.out.println("Saisir une chaine
                composee de plus de trois");
            str = clavier.nextLine();
            tabStr = str.split(" ");
        } while (tabStr.length < 3);
```

```
        for (int i = 0; i < 3; i++)
            acronyme += tabStr[i].charAt(0);

        System.out.println("Acronyme : " + acronyme.
            toUpperCase());
        clavier.close(); //fermer le clavier
    }
}
```