

Table des matières

1. TABLEAUX D'OBJETS.....	2
INTRODUCTION	2
PASSER DES OBJETS À DES MÉTHODES.....	4
COPIE D'UN OBJET ET ÉGALITÉ ENTRE DEUX OBJETS.....	6
2. CHAÎNES DE CARACTÈRES	9
INTRODUCTION	9
CRÉATION D'UN OBJET STRING	9
PRINCIPALES MÉTHODES DE LA CLASSE STRING	10
LA CLASSE STRINGTOKENIZER.....	12
EXEMPLE DE LECTURE DE FICHIER AVEC STRINGTOKENIZER.....	13
3. RECHERCHE DANS UN TABLEAU	14
RECHERCHE SÉQUENTIELLE DANS UN TABLEAU NON TRIÉ	14
RECHERCHE SÉQUENTIELLE DANS UN TABLEAU TRIÉ.....	15
RECHERCHE BINAIRE (DICHOTOMIQUE) DANS UN TABLEAU TRIÉ	17
4. TRI BULLE.....	19
5. INSERTION ET EFFACEMENT DANS UN TABLEAU TRIÉ	22
6. TABLEAUX À DEUX DIMENSIONS	23
7. CLASSES D'EMBALLAGE DE TYPE POUR LES TYPES PRIMITIFS	25

1. Tableaux d'objets

Introduction

Pour déclarer un tableau d'objets, il suffit de définir la classe de l'objet. Par exemple, soit une classe **Employe** déjà existante avec ses attributs et ses méthodes, on peut alors déclarer un tableau d'employés comme suit:

```
Employe tabEmpl[] = new Employe[7];
```

Cette déclaration a réservé en mémoire 7 emplacements pour recevoir 7 références à des objets de la classe **Employe**, mais les objets eux-mêmes n'ont pas encore été créés.

tabEmpl[0]	tabEmpl[1]	tabEmpl[2]	tabEmpl[3]	tabEmpl[4]	tabEmpl[5]	tabEmpl[6]

Voici un exemple qui crée une classe **Employe** dont les attributs sont le *numéro d'employé*, le *taux horaire* (fixé à 6.50) et le *nombre d'heures*. La classe **TestEmployes** va créer un tableau d'employés, pour ensuite afficher chaque employé avec son numéro, son taux, ses heures et son salaire brut.

```
// Fichier Employe.java
// Définition de la classe Employe
public class Employe
{
    private int numero;
    private double taux = 6.50;
    private double heures;

    public Employe(int num, double hres)
    {
        numero = num;
        heures = hres;
    }

    public double calculerBrut()
    {
        return taux * heures;
    }

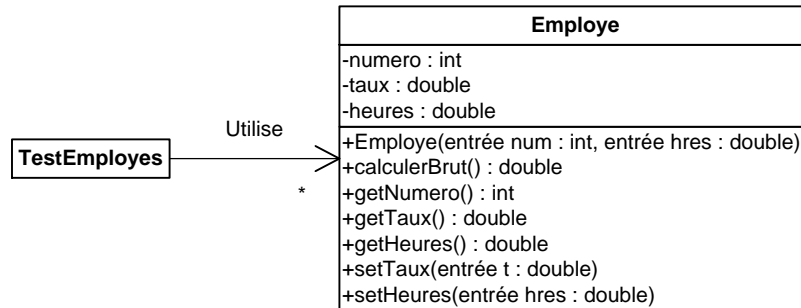
    public int getNumero()           { return numero; }

    public double getTaux()           { return taux; }

    public double getHeures()         { return heures; }

    public void setTaux(double t)     { taux = t; }

    public void setHeures(double hres) { heures = hres; }
}
```



```

// Fichier TestEmployes.java
// Création et utilisation d'un tableau d'Employe
import java.text.*;
public class TestEmployes
{
    public static void main(String args[])
    {
        DecimalFormat deuxDec = new DecimalFormat("0.00");
        final int NB_EMPL = 7;
        Employee tabEmpl[] = new Employee[NB_EMPL];

        // Construction des objets Employe
        for (int k = 0; k < NB_EMPL; k++)
            tabEmpl[k] = new Employee(101 + k, 40.0);

        // Modification du taux horaire de l'employé #106
        tabEmpl[5].setTaux(7.0);

        // Mettre les heures de l'employé #104 à 38

        // Affichage des informations
        System.out.println("*****Informations des employés*****");
        System.out.println("Numéro\t\tTaux\t\tHeures\t\tSalaire");

        for (int k = 0; k < NB_EMPL; k++)
            System.out.println(tabEmpl[k].getNumero() + "\t\t" +
                               deuxDec.format(tabEmpl[k].getTaux()) + "\t\t" +
                               deuxDec.format(tabEmpl[k].getHeures()) + "\t\t" +
                               deuxDec.format(tabEmpl[k].calculerBrut()));
    }
}
  
```

Résultats de l'exécution :

```

*****Informations des employés*****
Numéro      Taux      Heures      Salaire
101          6,50      40,00      260,00
102          6,50      40,00      260,00
103          6,50      40,00      260,00
104          6,50      38,00      247,00
105          6,50      40,00      260,00
106          7,00      40,00      280,00
107          6,50      40,00      260,00
  
```

Complétez la définition de la méthode qui recherche la position d'un numéro d'employé dans un tableau d'employés. Le tableau n'est pas nécessairement par ordre de numéro d'employé :

```
public static int rechercher(int noEmpl, Employe tabEmpl[])
{
    for (int k = 0; k < tabEmpl.length; k++)
        if (_____ )
            return k;

    return -1;
}
```

Passer des objets à des méthodes

Comme on peut passer des paramètres de type primitif à des méthodes, on peut aussi passer des paramètres de type objet à des méthodes. L'exemple suivant passe l'objet *monCercle* à la méthode *afficherCercle* :

```
public class TestParametre
{
    public static void main(String args[])
    {
        Cercle monCercle = new Cercle(5); // crée un objet Cercle de rayon 5
        afficherCercle(monCercle);
    }

    public static void afficherCercle(Cercle c)
    {
        System.out.println("La surface d'un cercle de rayon " + c.getRayon() +
                           " est de " + c.calculerAire());
    }
}
```

Il y a d'importantes différences entre passer une valeur de variable de type primitif et passer un objet :

- Passer une variable de type primitif signifie que c'est la valeur de la variable qui est passée au paramètre formel qui lui est associé. Changer la valeur du paramètre formel (considéré comme variable locale) à l'intérieur de la méthode n'affectera pas la valeur de la variable passée en paramètre. Ce passage de paramètres est appelé **passage de paramètre par valeur**.
- Passer un objet à une méthode signifie que c'est la référence à l'objet qui est passée au paramètre formel qui lui est associé. N'importe quel changement dans les attributs de l'objet local qui se produit à l'intérieur de la méthode va affecter l'objet original passé en paramètre. Dans la terminologie de la programmation, on dit que c'est un **passage de paramètres par référence**.

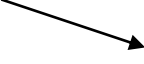
On retrouvera, à la page suivante, un exemple pour illustrer ces différences.

```
public class TestParametre
{
    public static void main(String args[])
    {
        Cercle monCercle = new Cercle(); // crée un objet Cercle de rayon 1
        int nbFois = 5;

        afficherDesCercles(monCercle, nbFois);

        System.out.println("\nrayon de monCercle est " + monCercle.getRayon());
        System.out.println("nbFois vaut " + nbFois);
    }

    public static void afficherDesCercles(Cercle c, int n)
    {
        System.out.println("Rayon\t\tSurface");
        while (n >= 1)
        {
            System.out.println(c.getRayon() + "\t\t" + c.calculerAire());
            c.setRayon(c.getRayon() + 1);
            n--;
        }
    }
}
```



on ajoute 1 à la valeur actuelle
du rayon du cercle

Résultats de l'exécution :

Rayon	Surface
1	3.14159
2	12.56636
3	28.27431
4	50.26544
5	78.53975

rayon de monCercle est 6
nbFois vaut 5

Copie d'un objet et égalité entre deux objets

Pour avoir un duplicata (copie) d'un objet existant, il s'agit d'offrir un constructeur dont le seul paramètre est une référence à un objet de la même classe à laquelle appartient le constructeur. On appelle alors ce constructeur le *constructeur copie*.

Les opérateurs d'égalité `==` et `!=`, lorsqu'ils sont utilisés avec des objets, ne produisent pas l'effet auquel on peut s'attendre. Au lieu de vérifier si un objet a les mêmes valeurs d'attributs que celles d'un autre objet, ces opérateurs déterminent plutôt si les deux objets sont le même objet, c'est-à-dire si les deux références contiennent la même adresse mémoire. Donc pour comparer des instances d'une classe et obtenir des résultats vraiment exploitables, vous devez implémenter des méthodes spéciales d'égalité dans votre classe.

Voici un exemple qui illustre le constructeur copie et l'implémentation d'une méthode d'égalité :

```
// Fichier Point.java
// Définition de la classe Point dont les objets représentent des points dans
// le plan cartésien
public class Point
{
    private int x;           // abscisse du point
    private int y;           // ordonnée du point

    public Point(int abs, int ord)
    {
        x = abs;
        y = ord;
    }

    public Point(Point p)      // le constructeur copie
    {
        x = p.x;
        y = p.y;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    public boolean equals(Point p)
    {
        return (x == p.x && y == p.y) ;
    }

    public String toString()
    {
        return "[" + x + ", " + y + "]";
    }
}
```

C'est comme définir une méthode `compareTo` pour l'objet en entier : retourne `true` si les *attributs* des objets ont les mêmes valeurs

```
// Fichier TestPoint.java
public class TestPoint
{
    public static void main(String args[])
    {
        Point point1 = new Point(2, 3);
        Point point2 = new Point(point1);

        System.out.println("point1 = " + point1);
        System.out.println("point2 = " + point2);

        if (point2.equals(point1))
            System.out.print("\npoint2 égale point1");
        else
            System.out.print("\npoint2 n'égale pas point1");

        if (point2 == point1)
            System.out.println(", mais point2 == point1");
        else
            System.out.println(", mais point2 != point1");
    }
}
```

point1 et point2 sont des références à deux objets différents dont les attributs ont les mêmes valeurs

`equals` compare les valeurs des attributs

`==` compare les références (adresses)

Résultats de l'exécution :

```
point 1 = [2, 3]
point 2 = [2, 3]

point2 égale point1, mais point2 != point1
```

L'implémentation locale de la méthode `equals()` garantit que `point1.equals(point2)` ne sera pas faux à moins que les deux objets `Point` représentent réellement deux points différents. Sans cette implémentation locale, l'expression `point1.equals(point2)` aurait invoqué la méthode `equals()` de la classe mère `Object`, laquelle aurait retourné la valeur `false` si les objets `Point` étaient distincts mais égaux. La méthode `equals()` que nous avons implémentée ne vient pas cependant redéfinir la méthode `equals()` de la classe `Object` car leurs signatures sont différentes. En effet, celle de la classe `Object` nécessite un paramètre de la classe `Object`.

Voici donc la façon correcte de redéfinir cette méthode, et, dans la classe Point, nous avons aussi redéfini la méthode `clone()` de la classe Object qui offre une autre façon d'obtenir une copie d'un objet sans avoir à faire un constructeur copie.

```
// Fichier Point.java
public class Point
{
    private int x;           // abscisse du point
    private int y;           // ordonnée du point

    public Point(int abs, int ord)
    {
        x = abs;
        y = ord;
    }

    public Object clone()      // redéfinition de la méthode clone()
    {
        return new Point(x, y);
    }

    public int getX() { return x; }
    public int getY() { return y; }

    public boolean equals(Object p)
    {
        if (p instanceof Point)
            return (x == ((Point)p).x && y == ((Point)p).y);
        else
            return false;
    }

    public String toString()
    {
        return "[" + x + ", " + y + "]";
    }
}
```

vérifie si p est bien un objet de la classe Point

```
// Fichier TestPoint.java
public class TestPoint
{
    public static void main(String args[])
    {
        Point point1 = new Point(2, 3);
        Point point2 = (Point)point1.clone();

        System.out.println("point1 = " + point1);
        System.out.println("point2 = " + point2);

        if (point2.equals(point1))
            System.out.print("\npoint2 égale point1");
        else
            System.out.print("\npoint2 n'égale pas point1");

        if (point2 == point1)
            System.out.println(", mais point2 == point1");
        else
            System.out.println(", mais point2 != point1");
    }
}
```

(Point) : demande explicite de transformer en type Point un objet d'une autre classe

2. Chaînes de caractères

Introduction

Les chaînes de caractères sont représentées en Java par les classes `String` ou `StringBuffer`. Ces classes sont définies dans `java.lang` et peuvent être utilisées dans n'importe quel programme sans les importer.

Les objets `String` représentent des chaînes constantes et les objets `StringBuffer` sont des chaînes modifiables. Java effectue cette distinction entre chaînes constantes ou modifiables pour des raisons de pure optimisation; en particulier, Java s'autorise certaines optimisations portant sur des objets `String`, telles que le partage d'un même objet `String` parmi plusieurs références, parce qu'il sait que ces objets ne changeront pas.

Lorsque vous avez le choix d'utilisation entre un objet `String` et un objet `StringBuffer` pour représenter une chaîne de caractères, préférez toujours l'objet `String` si vous êtes certain que cette chaîne de caractères ne changera pas. Ce choix améliore considérablement les performances.

Création d'un objet `String`

Une chaîne est un objet qu'on peut initialiser soit par initialisation, soit à l'aide de l'opérateur `new`.

En fait, la classe `String` fournit 9 constructeurs pour créer des objets `String`.

Exemples :

```
String chaine1 = "Bonjour";           // Bonjour

String chaine2 = new String("Bonjour"); // Bonjour

char tableauCar[] = {'S', 'a', 'l', 'u', 't'};
String chaine3 = new String(tableauCar); // Salut

String chaine4 = new String(tableauCar, 1, 3); // alu
```

Principales méthodes de la classe String

Les méthodes du tableau suivant (sauf la dernière) sont des méthodes d'instance. Il faut donc créer un objet String, par exemple *chaine*, pour pouvoir appeler ces méthodes d'instance. L'appel d'une de ces méthodes, appliquée à l'objet *chaine*, se fera à l'aide de :

`chaine.nomMéthode(arguments)`

Nom de la méthode		But de la méthode
char	charAt (int ind)	retourne le caractère de la chaîne à la position <i>ind</i>
int	length ()	retourne la longueur de la chaîne
Méthodes de comparaisons de chaînes		
int	compareTo (String autre)	compare <i>chaine</i> avec <i>autre</i> et renvoie une valeur négative, nulle ou positive suivant que <i>chaine</i> est respectivement plus petite, égale ou plus grande que <i>autre</i>
boolean	equals (Objet unObjet)	compare <i>chaine</i> à <i>unObjet</i> et renvoie <i>true</i> si <i>unObjet</i> est de la classe String et que les deux chaînes sont les mêmes
boolean	equalsIgnoreCase (String autre)	compare les chaînes en ignorant la différence entre minuscules et majuscules
boolean	startsWith (String autre)	teste si <i>chaine</i> débute avec la chaîne <i>autre</i>
boolean	startsWith (String autre, int ind)	teste si <i>chaine</i> débute avec la chaîne <i>autre</i> à partir de la position <i>ind</i>
boolean	endsWith (String autre)	teste si <i>chaine</i> se termine par la chaîne <i>autre</i>
Recherche de caractères et de sous-chaînes		
int	indexOf (char ch)	renvoie la position de la première occurrence de <i>ch</i> dans <i>chaine</i> ou -1 si pas trouvé
int	indexOf (char ch, int ind)	renvoie la position de la première occurrence de <i>ch</i> dans <i>chaine</i> à partir de <i>ind</i> ou -1 si pas trouvé
int	indexOf (String autre)	renvoie la position de la première occurrence de <i>autre</i> dans <i>chaine</i> ou -1 si pas trouvé
int	indexOf (String autre, int ind)	renvoie la position de la première occurrence de <i>autre</i> dans <i>chaine</i> à partir de la position <i>ind</i> ou -1 si pas trouvé
int	lastIndexOf (char ch)	renvoie la position de la dernière occurrence de <i>ch</i> dans <i>chaine</i> ou -1 si pas trouvé
int	lastIndexOf (char ch, int ind)	renvoie la position de la dernière occurrence de <i>ch</i> dans <i>chaine</i> à partir de la position <i>ind</i> en reculant ou -1 si pas trouvé
int	lastIndexOf (String autre)	renvoie la position de la dernière occurrence de <i>autre</i> dans <i>chaine</i> ou -1 si pas trouvé
int	lastIndexOf (String autre, int ind)	renvoie la position de la dernière occurrence de <i>autre</i> dans <i>chaine</i> à partir de la position <i>ind</i> en reculant ou -1 si pas trouvé

Extraction de sous-chaînes de caractères	
String substring (int ind)	renvoie une nouvelle chaîne contenant une copie des caractères de <i>chaîne</i> à partir de la position <i>ind</i> jusqu'à la fin de <i>chaîne</i>
String substring (int ind1, int ind2)	renvoie une nouvelle chaîne contenant une copie des caractères de <i>chaîne</i> à partir de la position <i>ind1</i> jusqu'à la position (<i>ind2</i> - 1) incluse. La nouvelle chaîne contiendra (<i>ind2</i> - <i>ind1</i>) caractères
Méthodes diverses	
String replace (char ch, char car)	renvoie une nouvelle chaîne dont toutes les occurrences de <i>ch</i> dans <i>chaîne</i> sont remplacées par <i>car</i> . La chaîne originale demeure inchangée
String toLowerCase ()	renvoie une nouvelle chaîne comprenant les caractères de <i>chaîne</i> convertis en minuscules. La chaîne originale demeure inchangée
String toUpperCase ()	renvoie une nouvelle chaîne comprenant les caractères de <i>chaîne</i> convertis en majuscules. La chaîne originale demeure inchangée
String trim ()	renvoie une nouvelle chaîne privée des espaces présents au début et à la fin de <i>chaîne</i> . La chaîne originale demeure inchangée
String toString ()	retourne la chaîne originale
Méthode de classe	
static String valueOf (argument)	renvoie une chaîne représentant la valeur de l'argument, par exemple : str = String.valueOf(12) retourne la chaîne "12" et la place dans <i>str</i> str = String.valueOf('a') retourne la chaîne "a" et la place dans <i>str</i>

La Classe StringTokenizer

Jusqu'à maintenant, nous avons surtout travaillé avec des mots. Une chaîne de caractères (String) représente aussi une phrase. La classe **StringTokenizer** est très utile pour **décomposer une phrase en morceaux** (comme les mots par exemple). Le StringTokenizer utilise un **séparateur** (le caractère d'espace, par défaut) pour déterminer quels sont les morceaux.

Pour utiliser cette classe il faut ajouter au tout début du programme l'instruction :
import java.util.*;

On peut spécifier le ou les délimiteurs à utiliser pour séparer les morceaux. Il suffit de fournir un String contenant tous les caractères délimiteurs comme deuxième paramètre au constructeur StringTokenizer.

```
// Sera découpé selon l'espace seulement - on retrouvera 4 morceaux
StringTokenizer mots = new StringTokenizer("C'est une belle journée.");

// Sera découpé selon /, : ou l'espace - on retrouvera 5 morceaux
StringTokenizer morceaux = new StringTokenizer("24/2/2007 10:34", "/: ");

// Sera découpé selon \t ou \n - on retrouvera 3 morceaux
StringTokenizer parties = new StringTokenizer("un un\tdeux\ntrois", "\t\n");
```

L'exemple ci-dessous affiche tous les mots (un par ligne) dans une phrase :

```
/* Mots.java
 * Affiche les mots d'une phrase un sous l'autre
 */
import java.util.*;
public class Mots {
    public static void main(String[] args) {

        // déclaration des variables
        String texte = "Bonjour tout le monde!";
        StringTokenizer mots = new StringTokenizer(texte) ;

        // compte le nombre de mots de la phrase
        System.out.println("Il y a " + mots.countTokens() +
            " mots dans la phrase :");

        // traitement des données
        while (mots.hasMoreTokens())           // regarde s'il reste des
                                                // mots dans la phrase
            // passe au morceau suivant et affiche le mot
            System.out.println("- " + mots.nextToken());
    }
}
```

Résultats de l'exécution :

```
Il y a 4 mots dans la phrase :
- Bonjour
- tout
- le
- monde!
```

Exemple de lecture de fichier avec StringTokenizer

```

import java.io.*;
import java.text.*;
import javax.swing.*;
import java.awt.*;
import java.util.*;
public class ExempleLectureFichier
{
    public static void main(String args[]) throws IOException
    {
        String    uneLigne,
                  nom,
                  prenom;
        char      sexe;
        int        noDossier;
        double     tauxHoraire;

        DecimalFormat monnaie = new DecimalFormat("0.00 $");
        StringTokenizer st;
        JTextArea sortie = new JTextArea();
        sortie.setFont(new Font("Courier", Font.PLAIN, 12));
        sortie.setTabSize(20);

        BufferedReader fichier = new BufferedReader(
                                new FileReader("personnel.txt"));

        sortie.append("numéro\tprénom et nom\tsexe\ttaux horaire");

        uneLigne = fichier.readLine();

        while (uneLigne != null) {
            // extraction des informations de la ligne lue
            st = new StringTokenizer(uneLigne);
            noDossier = Integer.parseInt(st.nextToken());
            nom       = st.nextToken();
            prenom    = st.nextToken();
            sexe      = st.nextToken().charAt(0);
            tauxHoraire = Double.parseDouble(st.nextToken());

            sortie.append("\n" + noDossier + "\t" + prenom + " " +
                        nom + "\t" + sexe + "\t" +
                        monnaie.format(tauxHoraire));

            uneLigne = fichier.readLine();
        } // fin du while

        fichier.close();

        JOptionPane.showMessageDialog(null, sortie,
            "Liste du fichier", JOptionPane.PLAIN_MESSAGE);
        System.exit(0);
    }
} // fin de la classe ExempleLectureFichier

```

Contenu du fichier :

123	Côté Denis	M	12.50
222	St-Pierre Nathalie	F	14.75
333	Lavoie Pierre-Luc	M	8
444	Robert Antoine	M	115.20
555	Tremblay Julie	F	9.99

Liste du fichier

numéro	prénom et nom	sexe	taux horaire
123	Denis Côté	M	12,50 \$
222	Nathalie St-Pierre	F	14,75 \$
333	Pierre-Luc Lavoie	M	8,00 \$
444	Antoine Robert	M	115,20 \$
555	Julie Tremblay	F	9,99 \$

OK

3. Recherche dans un tableau

Recherche séquentielle dans un tableau non trié

Principe :

Parcourir le tableau jusqu'à la fin pour vérifier si l'élément est là ou pas. Sortir de la boucle aussitôt que l'élément est trouvé. Si l'élément n'est pas trouvé, retourner -1.

Légende :

i =	indice dans le tableau
trouve =	pour indiquer si l'élément a été trouvé ou pas
nbEl =	nombre d'éléments dans le tableau
posi =	position de l'élément dans le tableau si trouvé
tablo =	tableau d'éléments
no =	élément à chercher dans le tableau

Algorithme :

```
i = 0
posi = -1
trouve = faux
tant que (i < nbEl et pas trouve)
    si tablo[i] = no
        trouve = vrai
        posi = i
    sinon
        i = i + 1
    fin si
fin tant que
retourne posi
```

```
static int rechercher(int no, int tablo[], int nbEl)
{
    int posi = -1;
    boolean trouve = false;

    for (int i = 0; i < nbEl && !trouve; i++)
        if (tablo[i] == no)
        {
            trouve = true;
            posi = i;
        }

    return posi;
}
```

Recherche séquentielle dans un tableau trié

Principe :

Parcourir le tableau jusqu'à ce qu'on trouve l'élément ou qu'on dépasse la valeur cherchée. Si l'élément n'est pas trouvé, retourner -1.

Légende :

i = indice dans le tableau
trouve = pour indiquer si l'élément a été trouvé ou pas
nbEl = nombre d'éléments dans le tableau
posi = position de l'élément dans le tableau si trouvé
tablo = tableau d'éléments
no = élément à chercher dans le tableau

Algorithme :

```
i = 0
posi = -1
trouve = faux
tant que (i < nbEl et pas trouve)
    si tablo[i] = no
        trouve = vrai
        posi = i
    sinon si tablo[i] > no
        i = nbEl
    sinon
        i = i + 1
    fin si
fin tant que
retourne posi
```

```
static int rechercher(int no, int tablo[], int nbEl)
{
    int posi = -1;
    boolean trouve = false;

    for (int i = 0; i < nbEl && !trouve; i++)
        if (tablo[i] == no)
        {
            trouve = true;
            posi = i;
        }
        else if (tablo[i] > no)
            i = nbEl;

    return posi;
}
```

pour arrêter la boucle for,
on met nbEl dans i

on aurait pu mettre true dans trouve :
cela aurait eu le même effet !

Exemple :

```
1      /*
2      * Ce programme lit une valeur à rechercher dans un tableau.
3      * Il affiche un message si la valeur n'existe pas et
4      * sa position si la valeur existe.
5      */
6      import javax.swing.*;
7      public class Sequentielle
8      {
9          public static void main(String[] args)
10         {
11             final int NB_ELEMENTS = 10;
12             int tablo[] = {2, 3, 4, 5, 6, 7, 8, 9, 11, 14};
13
14             int posi,
15                 valeur;
16
17             valeur = Integer.parseInt(JOptionPane.showInputDialog
18                                     ("Veuillez entrer la valeur"));
19             posi = rechercheSequentielle(tablo, valeur, NB_ELEMENTS);
20
21             if (posi == -1)
22                 JOptionPane.showMessageDialog(null,
23                                     "La valeur " + valeur + " n'est pas dans le tableau");
24             else
25                 JOptionPane.showMessageDialog(null,
26                                     "La valeur " + valeur + " est à la position " + posi);
27
28             System.exit(0);
29         }
30
31         // Cette méthode reçoit en paramètre le tableau, l'élément recherché
32         // et le nombre d'éléments dans le tableau. Elle retourne la
33         // position de la valeur recherchée
34         static int rechercheSequentielle(int tablo[], int no, int nbEl)
35         {
36             int i = 0;
37             int posi = -1;
38             boolean trouve = false;
39
40             while (i < nbEl && !trouve)
41             {
42                 if (tablo[i] == no)
43                 {
44                     trouve = true;
45                     posi = i;
46                 }
47                 else if (tablo[i] > no)
48                     i = nbEl;
49                 else
50                     i = i + 1;
51             }
52
53             return posi;
54         }
55     }
```


Recherche binaire (dichotomique) dans un tableau trié**Principe :**

Consiste à diviser le tableau en deux parties, à vérifier dans quelle partie du tableau peut se trouver l'élément en comparant l'élément cherché avec l'élément du milieu du tableau et à subdiviser ainsi de suite jusqu'à ce que l'élément cherché soit isolé. Si l'élément n'est pas trouvé, retourner -1.

Légende :

trouve= pour indiquer si l'élément a été trouvé ou pas
nbEl= nombre d'éléments dans le tableau
posi = position de l'élément dans le tableau si trouvé
tablo = tableau d'éléments
no = élément à chercher dans le tableau
debut, fin = indices pour délimiter le tableau
milieu = indice de l'élément au milieu du tableau

Algorithme :

```
posi = -1
trouve = false
debut = 0
fin = nbEl - 1
tant que (debut <= fin et pas trouve)
    milieu = (debut + fin) / 2
    si tablo[milieu] < no
        debut = milieu + 1
    sinon si tablo[milieu] > no
        fin = milieu - 1
    sinon
        posi = milieu
        trouve = true
    fin si
fin tant que
retourne posi
```

Exemple :

```
1      /*
2      * Ce programme lit une valeur à rechercher dans un tableau
3      * Il affiche un message si la valeur n'existe pas et
4      * sa position si la valeur existe.
5      */
6      import javax.swing.*;
7      public class Binaire
8      {
9          public static void main(String[] args)
10         {
11             final int NB_ELEMENTS = 10;
12             int tablo[] = {2, 3, 4, 5, 6, 7, 8, 9, 11, 14};
13
14             int posi,
15                 valeur;
16
17             valeur = Integer.parseInt(JOptionPane.showInputDialog
18                                     ("Veuillez entrer la valeur"));
19             posi = rechercheBinaire(tablo, valeur, NB_ELEMENTS);
20
21             if (posi == -1){
22                 JOptionPane.showMessageDialog(null,
23                                     "La valeur " + valeur + " n'est pas dans le tableau");
24             }
25             else
26                 JOptionPane.showMessageDialog(null,
27                                     "La valeur " + valeur + " est à la position " + posi);
28
29             System.exit(0);
30         }
31
32         // Cette méthode reçoit en paramètre le tableau, l'élément recherché
33         // et le nombre d'éléments contenus dans le tableau. Elle retourne la
34         // position de la valeur recherchée
35         static int rechercheBinaire(int tablo[], int no, int nbEl)
36         {
37             int debut = 0,
38                 posi = -1,
39                 fin = nbEl - 1,
40                 milieu;
41             boolean trouve = false;
42
43             while (debut <= fin && !trouve)
44             {
45                 milieu = (debut + fin) / 2;
46                 if (tablo[milieu] < no)
47                     debut = milieu + 1;
48                 else if (tablo[milieu] > no)
49                     fin = milieu - 1;
50                 else
51                 {
52                     posi = milieu;
53                     trouve = true;
54                 }
55             }
56             return posi;
57         }
58     }
59 }
```

4. Tri bulle

Principe :

Technique permettant de trier un tableau en ordre croissant en effectuant des permutations.
Pour nbEl éléments, la technique demande nbEl - 1 parcours.

Algorithme :

```
limite = nbEl - 1
faire
    pour (i = 0; i < limite; i++)
        si tablo[i] > tablo[i + 1]
            tempo = tablo[i]
            tablo[i] = tablo[i + 1]
            tablo[i + 1] = tempo
        fin si
    fin pour
    limite = limite - 1
tant que (limite > 0)
```

Exemple :

```
// Cette méthode permet de trier un tableau d'entiers de nbEl éléments
// version avec nbEl - 1 parcours
static void triBulle(int tablo[], int nbEl)
{
    int tempo;
    int limite = nbEl - 1;
    do
    {
        for (int i = 0; i < limite; i++)
            if (tablo[i] > tablo[i + 1])
            {
                tempo = tablo[i];
                tablo[i] = tablo[i + 1];
                tablo[i + 1] = tempo;
            }
        limite--;
    } while (limite > 0);
}
```

Ceci constitue une **permutation**, c'est-à-dire un échange de place entre deux éléments du tableau

Contenu du tableau :

Avant le tri					
5	2	8	1	9	12

Après le parcours 1					
2	5	1	8	9	12

Après le parcours 2					
2	1	5	8	9	12

Après le parcours 3					
1	2	5	8	9	12

Après le parcours 4					
1	2	5	8	9	12

Après le parcours 5					
1	2	5	8	9	12

Raffinement no 1 :

Pour réduire le nombre de parcours, on peut ajouter une variable booléenne pour savoir si oui ou non on a effectué une permutation dans un parcours. Si on n'a fait aucune permutation dans un parcours, c'est que le tableau est déjà trié et qu'il ne sert à rien de faire d'autres parcours.

Algorithme :

```
limite = nbEl - 1
faire
    trier = vrai
    pour (i = 0; i < limite; i++)
        si tablo[i] > tablo[i + 1]
            tempo = tablo[i]
            tablo[i] = tablo[i + 1]
            tablo[i + 1] = tempo
            trier = faux
    fin si
fin pour
limite = limite - 1
tant que (limite > 0 et pas trier)
```

Raffinement no 2 :

Pour raffiner davantage notre technique, on peut stopper un parcours à la position de la dernière permutation du parcours précédent, les nombres suivants étant déjà en ordre.

Algorithme :

```
limite = nbEl - 1
posiPerm = 0
faire
    trier = vrai
    pour (indice = 0; indice < limite; indice++)
        si tablo[indice] > tablo[indice + 1]
            tempo = tablo[indice]
            tablo[indice] = tablo[indice + 1]
            tablo[indice + 1] = tempo
            trier = faux
            posiPerm = indice
    fin si
fin pour
limite = posiPerm
tant que (limite > 0 et pas trier)
```

Exemple final :

```
// Cette méthode permet de trier un tableau d'entiers de nbEl éléments
// on arrête un parcours à la dernière permutation du parcours précédent
// et on arrête le tri si on n'a fait aucune permutation dans un parcours
static void triBulle(int tablo[], int nbEl)
{
    int tempo;
    int limite = nbEl - 1;
    int posiPerm = 0;
    boolean trier;

    do
    {
        trier = true;
        for (int i = 0; i < limite; i++)
            if (tablo[i] > tablo[i + 1])
            {
                tempo = tablo[i];
                tablo[i] = tablo[i + 1];
                tablo[i + 1] = tempo;
                trier = false;
                posiPerm = i;
            }
        limite = posiPerm;
    } while (limite > 0 && !trier);
}
```

Contenu du tableau :

Avant le tri					
5	2	8	1	9	12

Après le parcours 1					
2	5	1	8	9	12

Après le parcours 2					
2	1	5	8	9	12

Après le parcours 3					
1	2	5	8	9	12

Pour trier en ordre décroissant :

Il suffit d'inverser le if :

```
if (tablo[i] < tablo[i + 1])
```

Pour trier sur autre chose qu'un nombre :

On utilise la méthode `compareTo`. Par exemple, pour trier en ordre alphabétique :

```
if (tabNom[i].compareTo(tabNom[i + 1]) > 0)
```

Pour trier un tableau d'objets :

Il faut trier le tableau sur un des attributs. Par exemple, pour trier sur l'attribut *nom* :

```
if (tabObj[i].getNom().compareTo(tabObj[i + 1].getNom()) > 0)
```

5. Insertion et effacement dans un tableau trié**Principe :**

Pour insérer un élément dans un tableau trié, il s'agit de déplacer les éléments qui lui sont supérieurs d'une place vers la fin du tableau et ensuite d'insérer le nouvel élément à la place libérée. À la fin, on augmentera de un le nombre d'éléments présents dans le tableau. On aura bien sûr vérifié qu'il y a de la place dans le tableau avant d'y insérer un nouvel élément.

Algorithme :

```
ind = nbEl - 1
trouvePlace = faux
tant que (ind >= 0 et pas trouvePlace)
    si tablo[ind] > nb
        tablo[ind + 1] = tablo[ind]
        ind = ind - 1
    sinon
        trouvePlace = vrai
fin si
fin tant que
tablo[ind + 1] = nb
nbEl = nbEl + 1
```

Principe :

Pour supprimer un élément d'un tableau trié, il s'agit de déplacer les éléments qui lui sont supérieurs d'une place vers le début du tableau, écrasant ainsi l'élément supprimé. À la fin, on diminuera de un le nombre d'éléments présents dans le tableau. On aura bien sûr vérifié que l'élément était bien dans le tableau avant d'essayer de le supprimer.

Algorithme :

```
// on suppose que l'on connaît déjà la position de l'élément à supprimer
pour (ind = position; ind < nbEl - 1; ind++)
    tablo[ind] = tablo[ind + 1]
fin pour
nbEl = nbEl - 1
```

6. Tableaux à deux dimensions

indice [0] →	indice [0] [0]	indice [0] [1]	indice [0] [2]
indice [1] →	indice [1] [0]	indice [1] [1]	indice [1] [2]
indice [2] →	indice [2] [0]	indice [2] [1]	indice [2] [2]
indice [3] →	indice [3] [0]	indice [3] [1]	indice [3] [2]
indice [4] →	indice [4] [0]	indice [4] [1]	indice [4] [2]

définition :

Un tableau à deux dimensions correspond à un tableau dont chaque élément (rangée) est lui-même un tableau à une dimension.

déclaration :

```
int tabDeuxDim[][] = new int[5][3];    OU
```

```
int[][] tabDeuxDim = new int[5][3];
```

réserve l'espace pour un tableau de 5 rangées et de 3 colonnes

comme `int` est un type primitif numérique, chaque élément est initialisé à 0

initialisation :

À partir de la déclaration :

```
int tabDeuxDim[][] = { {10,23,13}, {5,76,3}, {50,60,70}, {5,6,7}, {0,4,8} };
```

définit un tableau de 5 rangées et 3 colonnes ayant les valeurs suivantes :

10	23	13
5	76	3
50	60	70
5	6	7
0	4	8

ainsi `tabDeuxDim[1][2]` correspond à la valeur 3

À partir de la logique :

Pour remplir le tableau, on utilise des boucles imbriquées, chaque boucle faisant varier l'indice d'une dimension.

Si on remplit le tableau dans le désordre (à partir d'un fichier par exemple), on calcule les deux indices pour chaque élément.

utilisation :

Pour accéder à un élément quelconque du tableau, on donne les deux indices (rangée et colonne)

exemple : `tabDeuxDim[0][2]` correspond au 3^e élément de la première rangée

longueur :

Chaque dimension a sa longueur

exemples :

`tabDeuxDim.length` donne la première dimension (nombre de rangées)

`tabDeuxDim[1].length` donne le nombre de colonnes de la deuxième rangée

paramètre :

Pour envoyer le tableau entier en paramètre, on met simplement son nom

exemples :

dans l'appel de la méthode : `afficher(tabDeuxDim);`

dans la signature de la méthode : `afficher(int tab[][])`

Pour envoyer un tableau à une dimension correspondant à la rangée 2 du tableau *tabDeuxDim*, on met le nom suivi de l'indice de la rangée

exemples :

dans l'appel de la méthode : `calculer(tabDeuxDim[2]);`

dans la signature de la méthode : `calculer(int tab[])`

Pour envoyer un élément du tableau *tabDeuxDim*, on met le nom suivi de l'indice de la rangée et de l'indice de la colonne

exemples :

dans l'appel de la méthode : `traiter(tabDeuxDim[2][0]);`

dans la signature de la méthode : `traiter(int nb)`

calcul du total de la rangée r :

```
int total = 0;
for (int col = 0; col < tabDeuxDim[r].length; col++)
    total += tabDeuxDim[r][col];
System.out.println("Le total de la rangée " + r + " est " + total);
```

calcul du total de la colonne c :

```
int total = 0;
for (int rang = 0; rang < tabDeuxDim.length; rang++)
    total += tabDeuxDim[rang][c];
System.out.println("Le total de la colonne " + c + " est " + total);
```

calcul du total de tout le tableau :

```
int total = 0;
for (int rang = 0; rang < tabDeuxDim.length; rang++)
    for (int col = 0; col < tabDeuxDim[rang].length; col++)
        total += tabDeuxDim[rang][col];
System.out.println("Le total du tableau est " + total);
```


7. Classes d'emballage de type pour les types primitifs

Les données de type primitif ne sont pas utilisées comme objets en Java car, la manipulation d'objets nécessitant plus de ressources, la performance du langage en souffrirait. Cependant, plusieurs méthodes en Java requièrent l'utilisation d'objets comme paramètres. Java offre donc un moyen de convertir une donnée primitive en un objet, en se servant des classes d'emballage de type. Chaque type de donnée primitif possède une classe d'emballage de type. Ces classes se nomment `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` et `Boolean`. Ces classes sont dans la librairie `java.lang`. Elles permettent la manipulation des types de donnée primitifs comme si c'étaient des objets de la classe `Object`. La plupart des classes que nous développons ou réutilisons manipulent et partagent des objets.

Les classes d'emballage sont appelées ainsi car chacune encapsule un type primitif de façon à ce qu'une variable de type primitif puisse être représentée par un objet lorsque c'est nécessaire et fournit des constructeurs, des constantes et des méthodes de conversion.

Par exemple, si nous voulons convertir l'entier 5 en un objet, et le nombre double 5.2 en un objet :

```
Integer objetEntier = new Integer(5); // ou bien new Integer("5");
Double objetDouble = new Double(5.2); // ou bien new Double("5.0");
```

Chaque classe d'emballage (sauf `Character` et `Boolean`) contient les méthodes d'instance `byteValue`, `shortValue`, `intValue`, `longValue`, `floatValue` et `doubleValue` qui permettent de convertir l'objet en donnée de type primitif. Par exemple :

```
long nb = objetDouble.longValue();
```

permet de convertir l'objet `objetDouble` en une valeur de type `long`

```
int nb = objetEntier.intValue();
```

permet de convertir l'objet `objetEntier` en nombre entier

De plus, chaque classe d'emballage redéfinit les méthodes `toString` et `equals` qui sont définies dans la classe `Object`. Par exemple :

```
double d = 5.9;
Double objDouble = new Double(d);
String s = objDouble.toString();
```

permet de convertir un nombre double primitif en une chaîne de caractères. Nous aurions pu faire aussi `String s = String.valueOf(d);`

Le diagramme suivant illustre les 6 méthodes de conversion que l'on peut utiliser pour convertir entre le type `int` et les classes `Integer` et `String`. Des méthodes similaires existent pour les 7 autres classes d'emballage.

