Exercices Java2

ALGORITHMES A TRADUIRE EN JAVA

Algorithme
Calcul de la valeur absolue d'un nombre réel

Objectif : Ecrire un programme Java servant à calculer la valeur absolue d'un nombre réel x à partir de la définition de la valeur absolue. La valeur absolue du nombre réel x est le nombre réel |x| :

|x| = x , si x ≥ 0

|x| = -x si x < 0

Spécifications de l'algorithme :

```
lire( x );
si x ≥ 0 alors  écrire( '|x|  =', x)
sinon  écrire( '|x|  =', -x)
fsi
```

Implantation en Java

Ecrivez avec les deux instructions différentes "**if...else..**" et "**...?.. : ...**", le programme Java complet correspondant à l'affichage ci-dessous :

```
Entrez un nombre x = -45
  |x|  = 45
```

Proposition de squelette de classe Java à implanter :

```
class ApplicationValAbsolue {
    public static void main(String[ ] args) {
       ……..
    }
  }
```

La méthode **main** calcule et affiche la valeur absolue.

Algorithme

Algorithme de résolution de l'équation du second degré dans R.

Objectif : On souhaite écrire un programme Java de résolution dans R de l'équation du second degré : $Ax^2 + Bx + C = 0$

Il s'agit ici d'un algorithme très classique provenant du cours de mathématique des classes du secondaire. L'exercice consiste essentiellement en la traduction immédiate

Spécifications de l'algorithme :

Algorithme Equation
  Entrée: A, B, C ∈ **Réels**
  Sortie: X1 , X2 ∈ **Réels**
  Local: Δ ∈ **Réels**

début
 lire(A, B, C);
 Si A=0 **alors début**{A=0}
  Si B = 0 **alors**
   Si C = 0 **alors**
    écrire(R est solution)
   Sinon{C ≠ 0}
    écrire(pas de solution)
   Fsi
  Sinon {B ≠ 0}
   X1 ← C/B;
   écrire (X1)
  Fsi
 fin
 Sinon {A ≠ 0}**début**
  Δ ← B^2 - 4*A*C ;
  Si Δ < 0 **alors**
   écrire(pas de solution)
  Sinon {Δ ≥ 0}
   Si Δ = 0 **alors**
    X1 ← -B/(2*A);
    écrire (X1)
   Sinon{Δ ≠ 0}
    X1 ← (-B +√Δ)/(2*A);
    X2 ← (-B - √Δ)/(2*A);

écrire(X1 , X2)
Fsi
Fsi
fin
Fsi
FinEquation

Implantation en Java

Ecrivez le programme Java qui est la traduction immédiate de cet algorithme dans le corps de la méthode main.

Proposition de squelette de classe Java à implanter :

```
class ApplicationEqua2 {
    public static void main(String[ ] args) {
        ……..
    }
}
```

Conseil :

On utilisera la méthode **static** sqrt(double x) de la classe **Math** pour calculer la racine carré d'un nombre réel :

√Δ se traduira alors par : Math.sqrt(delta)

Algorithme
Calcul des nombres de Armstrong

Objectif : On dénomme nombre de Armstrong un entier naturel qui est égal à la somme des cubes des chiffres qui le composent.

***Exemple* :**

$153 = 1^3 + 5^3 + 3^3$

153 = 1 + 125 + 27, est un nombre de Armstrong.

Spécifications de l'algorithme :

On sait qu'il n'existe que 4 nombres de Armstrong, et qu'ils ont tous 3 chiffres (ils sont compris entre 100 et 500).
Si l'on qu'un tel nombre est écrit ijk (i chiffre des centaines, j chiffres des dizaines et k chiffres des unités), il suffit simplement d'envisager tous les nombres possibles en faisant varier les chiffres entre 0 et 9 et de tester si le nombre est de Armstrong.

Implantation en Java

Ecrivez le programme Java complet qui fournisse les 4 nombres de Armstrong :

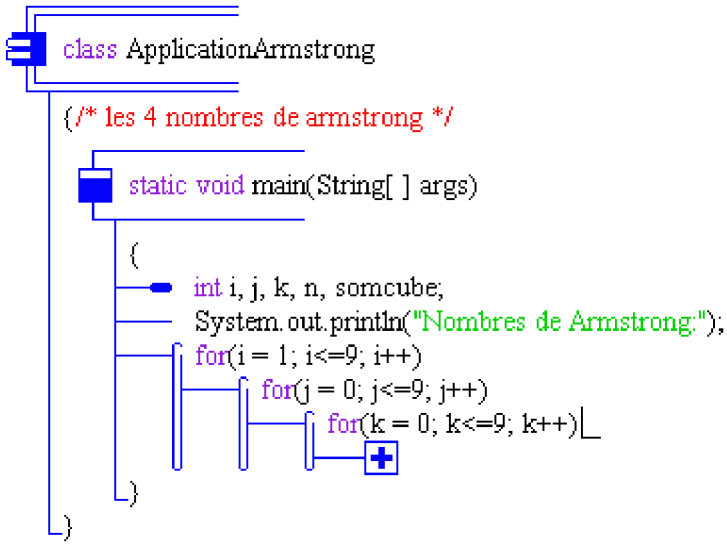
```
Nombres de Armstrong:
   153
   370
   371
   407
```

Proposition de squelette de classe Java à implanter :

```
class ApplicationArmstrong {
    public static void main(String[ ] args) {
        ……..
    }
}
```

La méthode **main** calcule et affiche les nombres de Armstrong.

Squelette plus détaillé de la classe Java à implanter :



Algorithme
Calcul de nombres parfaits

Objectif : On souhaite écrire un programme java de calcul des n premiers nombres parfaits. Un nombre est dit parfait s'il est égal à la somme de ses diviseurs, 1 compris.

***Exemple* : 6 = 1+2+3 , est un nombre parfait.**

Spécifications de l'algorithme :

l'algorithme retenu contiendra deux boucles imbriquées. Une boucle de comptage des nombres parfaits qui s'arrêtera lorsque le décompte sera atteint, la boucle interne ayant vocation à calculer tous les diviseurs du nombre examiné d'en faire la somme puis de tester l'égalité entre cette somme et le nombre.

```
Algorithme Parfait
  Entrée: n  ∈ N
  Sortie: nbr ∈ N
  Local: somdiv, k, compt   ∈ N

    début
      lire(n);
      compt  ← 0;
      nbr  ←2;
     Tantque(compt < n) Faire
       somdiv ← 1;
       Pour k ← 2 jusquà nbr-1 Faire
         Si reste(nbr par k) = 0 Alors // k  divise nbr
           somdiv ← somdiv + k
         Fsi
       Fpour ;
       Si somdiv = nbr Alors
         ecrire(nbr) ;
         compt ← compt+1;
       Fsi;
       nbr ← nbr+1
     Ftant
    FinParfait
```

Implantation en Java

Ecrivez le programme Java complet qui produise le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

```
Entrez combien de nombre parfaits : 4
    6 est un nombre parfait
    28 est un nombre parfait
    496 est un nombre parfait
    8128 est un nombre parfait
```

Proposition de squelette de classe Java à implanter :

```
class ApplicationParfaits {
    public static void main(String[ ] args) {
        ……..
    }
}
```

La méthode **main** calcule et affiche les nombres parfaits

Algorithme

Calcul du pgcd de 2 entiers (méthode Euclide)

Objectif : On souhaite écrire un programme de calcul du pgcd de deux entiers non nuls, en Java à partir de l'algorithme de la méthode d'Euclide. Voici une spécification de l'algorithme de calcul du PGCD de deux nombres (entiers strictement positifs) a et b, selon cette méthode :

Spécifications de l'algorithme :

```
Algorithme Pgcd
 Entrée: a,b  ∈  N* x N*
 Sortie: pgcd  ∈ N
 Local: r,t  ∈ N x N

début
  lire(a,b);
  Si ba Alors
    t ← a ;
    a ← b ;
    b ← t
  Fsi;
  Répéter
    r ← a mod b ;
    a ← b ;
    b ← r
  jusquà r = 0;
  pgcd ← a;
  ecrire(pgcd)
FinPgcd
```

Implantation en Java

Ecrivez le programme Java complet qui produise le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Entrez le premier nombre : *21*
Entrez le deuxième nombre : *45*
Le PGCD de 21 et 45 est : 3

Proposition de squelette de classe Java à implanter :

```
class ApplicationEuclide {
    public static void main(String[ ] args) {
        ……..
    }
    static int pgcd (int a, int b) {
        ……..
    }
}
```

La méthode **pgcd** renvoie le pgcd des deux entiers p et q .

Algorithme
Calcul du pgcd de 2 entiers (méthode Egyptienne)

Objectif : On souhaite écrire un programme de calcul du pgcd de deux entiers non nuls, en Java à partir de l'algorithme de la méthode dite "égyptienne " Voici une spécification de l'algorithme de calcul du PGCD de deux nombres (entiers strictement positifs) p et q, selon cette méthode :

Spécifications de l'algorithme :

```
Lire (p, q ) ;

Tantque p ≠ q faire
    Si p > q alors

      p ← p – q

    sinon

      q ← q – p

    FinSi
FinTant;
Ecrire(  " PGCD = " , p )
```

Implantation en Java

Ecrivez le programme Java complet qui produise le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Entrez le premier nombre : *21*
Entrez le deuxième nombre : *45*
Le PGCD de 21 et 45 est : 3

Proposition de squelette de classe Java à implanter :

```
class ApplicationEgyptien {
    public static void main(String[ ] args) {
        ……..
    }
    static int pgcd (int p, int q) {
        ……..
    }
}
```

La méthode **pgcd** renvoie le pgcd des deux entiers p et q .

Algorithme

Calcul de nombres premiers (boucles **while** et **do**…**while**)

Objectif : On souhaite écrire un programme Java de calcul et d'affichage des n premiers nombres premiers. Un nombre entier est premier s'il n'est divisible que par 1 et par lui-même On opérera une implantation avec des boucles while et do...while.

Exemple : 37 est un nombre premier

Spécifications de l'algorithme :

```
Algorithme Premier
  Entrée: n  ∈ N
  Sortie: nbr   ∈ N
  Local: Est_premier  ∈ {Vrai , Faux}
         divis,compt  ∈N²;

   début
    lire(n);
    compt  ← 1;
    ecrire(2);
    nbr ← 3;
    Tantque(compt < n) Faire
     divis ← 3;
     Est_premier ← Vrai;
     Répéter
       Si reste(nbr par divis) = 0 Alors
        Est_premier ← Faux
       Sinon
        divis ← divis+2
       Fsi
     jusquà (divis > nbr / 2)ou (Est_premier=Faux);
     Si Est_premier =Vrai Alors
       ecrire(nbr);
       compt ← compt+1
     Fsi;
     nbr ← nbr+1
    Ftant
  FinPremier
```

Implantation en Java

Ecrivez le programme Java complet qui produise le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Combien de nombres premiers : *5*
2
3
5
7
11

Proposition de squelette de classe Java à implanter avec une boucle while et une boucle do...while imbriquée :

On étudie la primalité de tous les nombres systématiquement

```
class ApplicationComptPremiers1 {
    public static void main(String[ ] args) {
      .....
      while (compt < n)
      {
          divis=2 ;
          Est_premier=true;
          do
          {
          }
          while(Est_premier);
          nbr++ ;
      }
      ......
    }
}
```

La méthode **main** affiche la liste des nombres premiers demandés.

Algorithme

Calcul de nombres premiers (boucles **for**)

Objectif : On souhaite écrire un programme Java de calcul et d'affichage des n premiers nombres premiers. Un nombre entier est premier s'il n'est divisible que par 1 et par lui-même. On opérera une implantation avec des boucles for imbriquées.

Exemple : 19 est un nombre premier

Spécifications de l'algorithme : (o*n étudie la primalité des nombres uniquement impairs)*

```
Algorithme Premier
  Entrée: n  ∈ N
  Sortie: nbr   ∈ N
  Local: Est_premier  ∈ {Vrai , Faux}
         divis,compt  ∈N²;

   début
    lire(n);
    compt  ← 1;
    ecrire(2);
    nbr ← 3;
    Tantque(compt < n) Faire
     divis ← 3;
     Est_premier ← Vrai;
     Répéter
       Si reste(nbr par divis) = 0 Alors
        Est_premier ← Faux
       Sinon
        divis ← divis+2
       Fsi
     jusquà (divis > nbr / 2)ou (Est_premier=Faux);
     Si Est_premier =Vrai Alors
       ecrire(nbr);
       compt ← compt+1
     Fsi;
     nbr ← nbr+2 // nbr impairs
    Ftant
  FinPremier
```

Implantation en Java

Ecrivez le programme Java complet qui produise le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Combien de nombres premiers : *5*
2
3
5
7
11

Proposition de squelette de classe Java à implanter avec deux boucles **for** imbriquées :

On étudie la primalité des nombres uniquement impairs

```
class ApplicationComptPremiers2 {
    public static void main(String[ ] args) {
        .....
  for( nbr = 3; compt < max; nbr += 2 )
  { Est_premier = true;
      for (divis = 2; divis<= nbr/2; divis++ )
          {
      if(Est_premier)
          {
  }
      ......
    }
}
```

La méthode **main** affiche la liste des nombres premiers demandés.

Le fait de n'étudier la primalité que des nombres impairs accélère la vitesse d'exécution du programme, il est possible d'améliorer encore cette vitesse en ne cherchant que les diviseurs dont le carré est inférieur au nombre (test : **jusquà** ($divis^2$ > nbr)**ou** (Est_premier=Faux))

Algorithme
Calcul du nombre d'or

Objectif : On souhaite écrire un programme Java qui calcule le nombre d'or utilisé par les anciens comme nombre idéal pour la sculpture et l'architecture. Si l'on considère deux suites numériques (U) et (V) telles que pour n strictement supérieur à 2 :

$$U_n = U_{n-1} + U_{n-2}$$

et

$$V_n = U_n / U_{n-1}$$

On montre que la suite (V) tend vers une limite appelée nombre d'or (nbr d'Or = 1,6180339887498948482045868343656 4).

Spécifications de l'algorithme :

n, U_n, U_{n1}, U_{n2} : sont des entiers naturels
V_n, V_{n1}, ε : sont des nombres réels

lire(ε); *// précision demandée*
  $U_{n2} \leftarrow 1$;
  $U_{n1} \leftarrow 2$;
  $V_{n1} \leftarrow 2$;
  $n \leftarrow 2$; *// rang du terme courant*
Itération
  $n \leftarrow n + 1$;
  $U_n \leftarrow U_{n1} + U_{n2}$;
  $V_n \leftarrow U_n / U_{n1}$;
  si $|V_n - V_{n1}| \leq \varepsilon$ **alors** Arrêt de la boucle ; *// la précision est atteinte*
  sinon
    $U_{n2} \leftarrow U_{n1}$;
    $U_{n1} \leftarrow U_n$;
    $V_{n1} \leftarrow V_n$;
  fsi
fin Itération
ecrire (V_n , n);

Ecrire un programme fondé sur la spécification précédente de l'algorithme du calcul du nombre d'or. Ce programme donnera une valeur approchée avec une précision fixée de ε du

nombre d'or. Le programme indiquera en outre le rang du dernier terme de la suite correspondant.

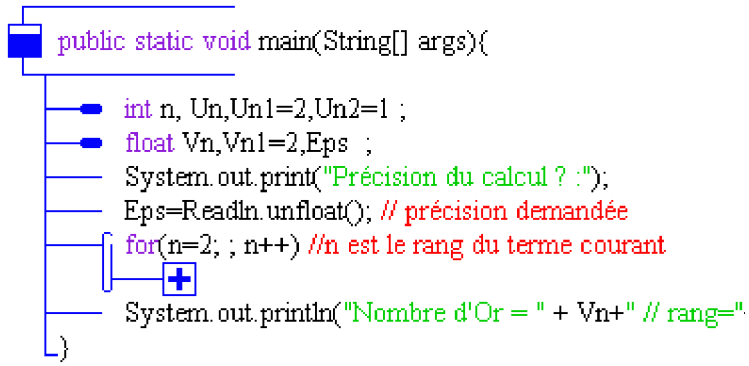
Implantation en Java

On entre au clavier un nombre réel ci-dessous 0.00001, pour la précision choisie (ici 5 chiffres après la virgule), puis le programme calcule et affiche le Nombre d'or (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Précision du calcul ? : *0.00001*
Nombre d'Or = 1.6180328 // rang=14

Proposition de squelette de classe Java à implanter avec un boucle for :

class AppliNombredOr **{**



}

Remarquons que nous proposons une boucle **for** ne contenant pas de condition de rebouclage dans son en-tête (donc en apparence infinie), puisque nous effectuerons le test "**si** |Vn - Vn1| <= Eps **alors** Arrêt de la boucle" qui permet l'arrêt de la boucle. Dans cette éventualité , la boucle **for** devra donc contenir dans son corps, une instruction de rupture de séquence.

Algorithme
Conjecture de Goldbach

Objectif : On souhaite écrire un programme Java afin de vérifier sur des exemples, la conjecture de GoldBach (1742), soit : "Tout nombre pair est décomposable en la somme de deux nombres premiers".

Dans cet exercice nous réutilisons un algorithme déjà traité (algorithme du test de la primalité d'un nombre entier), nous rappelons ci-après un algorithme indiquant si un entier "nbr" est premier ou non :

```
Algorithme Premier
  Entrée: nbr  ∈ N
  Local: Est_premier  ∈ {Vrai , Faux}
         divis,compt  ∈ N² ;

début
 lire(nbr);
 divis ← 3;
 Est_premier ← Vrai;
 Répéter
   Si reste(nbr par divis) = 0 Alors
    Est_premier ← Faux
   Sinon
    divis ← divis+2
   Fsi
 jusquà (divis > nbr / 2)ou (Est_premier=Faux);
 Si Est_premier = Vrai Alors
   ecrire(nbr est premier)
 Sinon
   ecrire(nbr n'est pas premier)
 Fsi
FinPremier
```

Conseil :

Il faudra traduire cet algorithme en fonction recevant comme paramètre d'entrée le nombre entier dont on teste la primalité, et renvoyant un booléen **true** ou **false** selon que le nombre entré a été trouvé ou non premier

Spécifications de l'algorithme de Goldbach :

En deux étapes :

1. On entre un nombre pair n au clavier, puis on génère tous les couples (a,b) tels que a + b = n, en faisant varier a de 1 à n/2. Si l'on rencontre un couple tel que a et b soient simultanément premiers la conjecture est vérifiée.

2. On peut alors, au choix soit arrêter le programme, soit continuer la recherche sur un autre nombre pair.

Exemple :
Pour n = 10, on génère les couples :
(1,9), (2,8), (3,7), (4,6), (5,5)
on constate que la conjecture est vérifiée, et on écrit :
10 = 3 + 7
10 = 5 + 5

Implantation en Java

On écrira la méthode booléenne **EstPremier** pour déterminer si un nombre est premier ou non, et la méthode **generCouples** qui génère les couples répondant à la conjecture.

Proposition de squelette de classe Java à implanter :

```
class ApplicationGoldBach  {
    public static void main(String[ ] args) {
        ……..
    }
    static boolean EstPremier(int m)  {
        ……..
    }
    static void generCouples(int n)  {
        ……..
    }
}
```

Algorithme
Méthodes d'opérations sur 8 bits

Objectif : On souhaite écrire une application de manipulation interne des bits d'une variable entière non signée sur 8 bits. Le but de l'exercice est de construire une famille de méthodes de travail sur un ou plusieurs bits d'une mémoire. L'application à construire contiendra 9 méthodes :

Spécifications des méthodes :

1. Une méthode **BitSET** permettant de mettre à **1** un bit de rang fixé.
2. Une méthode **BitCLR** permettant de mettre à **0** un bit de rang fixé.
3. Une méthode **BitCHG** permettant de remplacer un bit de rang fixé par son complément.
4. Une méthode **SetValBit** permettant de modifier un bit de rang fixé.
5. Une méthode **DecalageD** permettant de décaler les bits d'un entier, sur la droite de **n** positions (introduction de **n** zéros à gauche).
6. Une méthode **DecalageG** permettant de décaler les bits d'un entier, sur la gauche de **n** positions (introduction de **n** zéros à droite).
7. Une méthode **BitRang** renvoyant le bit de rang fixé d'un entier.
8. Une méthode **ROL** permettant de décaler avec rotation, les bits d'un entier, sur la droite de **n** positions (réintroduction à gauche).
9. Une méthode **ROR** permettant de décaler avec rotation, les bits d'un entier, sur la gauche de **n** positions (réintroduction à droite).

Exemples de résultats attendus :

Prenons une variable X entier (par exemple sur 8 bits) X = **11100010**

1. **BitSET** (X,3) = X = **11101010**
2. **BitCLR** (X,6) = X = **10100010**
3. **BitCHG** (X,3) = X = **11101010**
4. **SetValBit**(X,3,1) = X = **11101010**
5. **DecalageD** (X,3) = X = **00011100**
6. **DecalageG** (X,3) = X = **00010000**
7. **BitRang** (X,3) = 0 ; BitRang (X,6) = 1 ...
8. **ROL** (X,3) = X = **00010111**
9. **ROR** (X,3) = X = **01011100**

Implantation en Java

La conception de ces méthodes ne dépendant pas du nombre de bits de la mémoire à opérer on choisira le type int comme base pour une mémoire. Ces méthodes sont classiquement des outils de manipulation de l'information au niveau du bit.

Lors des jeux de tests pour des raisons de simplicité de lecture il est conseillé de ne rentrer que des valeurs entières portant sur 8 bits.

Il est bien de se rappeler que le type primaire **int** est un type entier signé sur 32 bits (représentation en complément à deux).

Proposition de squelette de classe Java à implanter :

```
class Application8Bits {
    public static void main(String [ ] args){
        ……..       }
     static int BitSET (int nbr, int num) {  ……..      }
     static int BitCLR (int nbr, int num) {  ……..      }
     static int BitCHG (int nbr, int num) {  ……..      }
     static int SetValBit (int nbr, int rang, int val) {  ……..      }
     static int DecalageD (int nbr, int n) {  ……..      }
     static int DecalageG (int nbr, int n) {  ……..      }
     static int BitRang (int nbr, int rang) {  ……..   }
     static int ROL (int nbr, int n) {   ……..    }
     static int ROR (int nbr, int n) {  ……..    }
  }
```

SOLUTIONS DES ALGORITHMES EN JAVA

Classe Java solution

Calcul de la valeur absolue d'un nombre réel

Objectif : Ecrire un programme Java servant à calculer la valeur absolue d'un nombre réel x à partir de la définition de la valeur absolue. La valeur absolue du nombre réel x est le nombre réel |x| :

|x| = x , si x ≥ 0

|x| = -x si x < 0

Spécifications de l'algorithme :

lire(x);
si x>=0 **alors** écrire('|x| =', x)
sinon écrire('|x| =', -x)
fsi

Implantation en Java avec un if…else :

```
class ApplicationValAbsolue
{
   static void main(String[ ] args)  {
      float x;
      System.out.print("Entrez un nombre x = ");
      x = Readln.unfloat();
      if (x<0)
         System.out.println("|x| = "+(-x));
      else
         System.out.println("|x| = "+x);
   }
}
```

Implantation en Java avec un "... ? ... : ..." :

```
class ApplicationValAbsolue {
   static void main(String[ ] args) {
      float x;
      System.out.print("Entrez un nombre x = ");
      x = Readln.unfloat();
      System.out.println("|x| = "+ (x<0 ? -x : x) );
   }
}
```

Classe Java solution

Algorithme de résolution de l'équation du second degré dans R.

Objectif : Ecrire un programme Java de résolution dans R de l'équation du second degré : $Ax^2 + Bx + C = 0$

Implantation en Java de la classe

```
class ApplicationEqua2 {
    static void main (String [] arg)    {
    }
}
```

Implantation en Java de la méthode main :

```
static void main (String [] arg)    {
    double a, b, c, delta ;
    double x, x1, x2 ;
    System.out.print("Entrer une valeur pour a : ") ;
    a =  Readln.undouble() ;
    System.out.print("Entrer une valeur pour b : ") ;
    b =  Readln.undouble() ;
    System.out.print("Entrer une valeur pour c : ") ;
    c =  Readln.undouble() ;
    if (a ==0)  {
        if (b ==0)  {
            if (c ==0)    {
                System.out.println("tout reel est solution") ;
            }
            else  {
                System.out.println("il n'y a pas de solution") ;
            }
        }
        else  {
            x = -c/b ;
            System.out.println("la solution est " + x) ;
        }
    }
    else  {
        delta = b*b -4*a*c ;
        if (delta <0)     {
            System.out.println("il n'y a pas de solution dans les reels") ;
        }
        else  {
            x1 = (-b + Math.sqrt(delta))/ (2*a) ;
            x2 = (-b - Math.sqrt(delta))/ (2*a) ;
            System.out.println("il y deux solutions egales a " + x1 + " et " + x2) ;
        }
    }
}
```

Classe Java solution
Calcul des nombres de Armstrong

Objectif : On dénomme nombre de Armstrong un entier naturel qui est égal à la somme des cubes des chiffres qui le composent. Ecrire un programme Java qui affiche de tels nombres.

***Exemple* :**

$153 = 1^3 + 5^3 + 3^3$

153 = 1 + 125 + 27, est un nombre de Armstrong.

Implantation en Java

```
class ApplicationArmstrong {
    static void main(String[ ] args) {
        int i, j, k, n, somcube;
        System.out.println("Nombres de Armstrong:");
        for(i = 1; i<=9; i++)
            for(j = 0; j<=9; j++)
                for(k = 0; k<=9; k++)
                { n = 100*i + 10*j + k;
                  somcube = i*i*i + j*j*j + k*k*k;
                  if (somcube == n) System.out.println(n);
                }
    }
}
```

Classe Java solution

Calcul de nombres parfaits

Objectif : On souhaite écrire un programme java de calcul des n premiers nombres parfaits. Un nombre est dit parfait s'il est égal à la somme de ses diviseurs, 1 compris. Ecrire un programme Java qui affiche de tels nombres.

***Exemple* : 6 = 1+2+3 , est un nombre parfait.**

Implantation en Java

Ci-dessous le programme Java complet qui produit le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Entrez combien de nombre parfaits : *4*
6 est un nombre parfait
28 est un nombre parfait
496 est un nombre parfait
8128 est un nombre parfait

```
class ApplicationParfaits {
   static void main(String[ ] args) {
      int compt = 0, n, k, somdiv, nbr;
      System.out.print("Entrez combien de nombre parfaits : ");
      n = Readln.unint( );
      nbr = 2;
      while (compt != n)
      { somdiv = 1;
        k = 2;
        while(k <= nbr/2 )
        {
          if (nbr % k == 0) somdiv += k ;
          k++;
        }
        if (somdiv == nbr)
        { System.out.println(nbr+" est un nombre parfait");
          compt++;
        }
        nbr++;
      }
   }
}
```

Classe Java solution

Calcul du pgcd de 2 entiers (méthode Euclide)

Objectif : Ecrire un programme de calcul du pgcd de deux entiers non nuls, en Java à partir de l'algorithme de la méthode d'Euclide.

Implantation en Java

Ci-dessous le programme Java complet qui produit le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Entrez le premier nombre : *21*
Entrez le deuxième nombre : *45*
Le PGCD de 21 et 45 est : 3

```
                    // PGCD - EUCLIDE  avec des méthodes
class TD2ApplicationPgcdEuclide{

   static int Pgcd(int a, int b)
   { // renvoie le pgcd de a>=b
      int r;
      do {
         r = a % b;
         a = b;
         b = r;
      }while(r !=0);
      return a;
   }

   static int Max(int a, int b)
   { // renvoie le max de a et de b
      return a>b ? a : b;
   }

   static int Min(int a, int b)
   { // renvoie le min de a et de b
      return a>b ? b : a;
   }

   public static void main(String[ ] args)
   {
      int a = Readln.unint();
      int b = Readln.unint();
      int max = Max(a,b), min = Min(a,b);
      a = max;
      b = min;
      if(min !=0)
         System.out.println("pgcd de "+a+" et de "+b+" = "+Pgcd(a,b));
      else System.out.println("calcul impossible");
   }
}
```

Classe Java solution

Calcul du pgcd de 2 entiers (méthode Egyptienne)

Objectif : Ecrire un programme de calcul du pgcd de deux entiers non nuls, en Java à partir de l'algorithme de la méthode dite "égyptienne ".

Implantation en Java

Ci-dessous le programme Java complet qui produit le dialogue suivant à l'écran (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Entrez le premier nombre : *21*
Entrez le deuxième nombre : *45*
Le PGCD de 21 et 45 est : 3

```
class ApplicationEgyptien {

  static void main(String[ ] args) {
    int p,q;
    System.out.print("Entrez le premier nombre : ");
    p = Readln.unint( );// méthode permettant de lire un entier au clavier
    System.out.print("Entrez le deuxième nombre : ");
    q = Readln.unint( );
    if (p*q!=0)
       System.out.println("Le pgcd de "+p+" et de "+q+" est "+pgcd(p,q));
    else
      System.out.println("Le pgcd n'existe pas lorsque l'un des deux nombres est nul !");
  }


  static int pgcd (int p, int q) {
    while ( p != q) {
      if (p>q) p -= q;
      else q -= p;
    }
    return p;
  }

}
```

Classe Java solution

Calcul de nombres premiers (boucles **while** et **do**…**while**)

Objectif : On souhaite écrire un programme Java de calcul et d'affichage des n premiers nombres premiers.

Implantation en Java avec une boucle while et une boucle do...while imbriquée

```
//une liste des n premiers nombres premiers
class ApplicationComptPremiers1
{
  static public void main(String[ ] args)
  {
    int divis, nbr, n, compt = 0 ;
    boolean Est_premier;
    System.out.print("Combien de nombres premiers :  ");
    n = Readln.unint();
    System.out.println( 2 );
    nbr=3;
    while (compt < n-1) {
      divis=2 ;
      Est_premier=true;
      do{
        if(nbr % divis ==0)   Est_premier=false;
        else divis = divis+1 ;
      }while ((divis <= nbr/2)&& (Est_premier==true));
      if(Est_premier) {
        compt++;
        System.out.println( nbr );
      }
      nbr++ ;
    }
  }
}
```

Ce programme Java produit le dialogue suivant à l'écran:

```
Combien de nombres premiers :  5
  2
  3
  5
  7
  11
```

Classe Java solution

Calcul de nombres premiers (boucles **for**)

Objectif : On souhaite écrire un programme Java de calcul et d'affichage des n premiers nombres premiers.

Implantation en Java avec deux boucles **for** imbriquées :

```
class ApplicationComptPremiers2 {

   static void main(String[ ] args) {
      int divis, nbr, n, compt = 1 ;
      boolean Est_premier;
      System.out.print("Combien de nombres premiers :  ");
      n = Readln.unint();
      System.out.println( 2 );
      for( nbr = 3; compt < n  ; nbr += 2 )
      { Est_premier = true;
         for (divis = 2; divis<= nbr/2; divis++ )
            if ( nbr % divis == 0 )
            { Est_premier = false;
               break;
            }
         if (Est_premier)
         {
            compt++;
            System.out.println( nbr );
         }
      }
   }
}
```

Le programme Java précédent produit le dialogue suivant à l'écran :

Combien de nombres premiers : *5*
2
3
5
7
11

Classe Java solution
Calcul du nombre d'or

Objectif : On souhaite écrire un programme Java qui calcule le nombre d'or utilisé par les anciens comme nombre idéal pour la sculpture et l'architecture (nbr d'Or = 1,6180339887498948482045868343656 4).

Implantation en Java avec un boucle for :

Le programme ci-dessous donne une valeur approchée avec une précision fixée de ε du nombre d'or. Le programme indique en outre le rang du dernier terme de la suite correspondant à la valeur approchée calculée.

Exemple : On entre au clavier un nombre réel ci-dessous 0.00001, pour la précision choisie (ici 5 chiffres après la virgule), puis le programme calcule et affiche le Nombre d'or (les caractères gras représentent ce qui est écrit par le programme, les italiques ce qui est entré au clavier) :

Précision du calcul ? : *0.00001*
Nombre d'Or = 1.6180328 // rang=14

```
class AppliNombredOr {

  static void main(String[ ] args) {
    int n, Un, Un1=2, Un2=1 ;
    float Vn,Vn1=2, Eps  ;
    System.out.print("Précision du calcul ? :");
    Eps=Readln.unfloat(); // précision demandée
    for (n=2; ; n++) //n est le rang du terme courant
    {
      Un = Un1 + Un2;
      Vn =(float)Un / (float)Un1;
      if (Math.abs(Vn - Vn1) <= Eps)
        break;
      else
      {
        Un2 = Un1;
        Un1 = Un;
        Vn1 = Vn;
      }
    }
    System.out.println("Nombre d'Or = " + Vn+" // rang="+n);
  }
}
```

Classe Java solution
Conjecture de Goldbach

Objectif : On souhaite écrire un programme Java afin de vérifier sur des exemples, la conjecture de GoldBach (1742), soit : "Tout nombre pair est décomposable en la somme de deux nombres premiers".

Implantation en Java :

```
class ApplicationGoldBach {

   static void main(String[ ] args) {
      int n;
      while ( (n=Readln.unint( )) !=0 ){
         generCouples(n); }
   }

   static boolean EstPremier(int m)  {
      int k ;
      for (k = 2 ; k <= m / 2 ; k++) {
         if (m % k == 0) {
            return  false;
         }
      }
      return true;
   }

   static void generCouples(int n) {
      if (n % 2 ==0) {
         for (int a = 1; a <= n/2; a++) {
            int  b;
            b = n - a;
            if ( EstPremier(a) && EstPremier(b) ) {
               System.out.println(n+" = "+a+" + "+b);
            }
         }
      }
      else System.out.println("Votre nombre n'est pas pair !");
   }
}
```

Classe Java solution
Méthodes d'opérations sur 8 bits

Objectif : On souhaite écrire une application de manipulation interne des bits d'une variable entière non signée sur 8 bits.

Implantation en Java

La conception de ces méthodes ne dépendant pas du nombre de bits de la mémoire à opérer on choisira le type int comme base pour une mémoire. Ces méthodes sont classiquement des outils de manipulation de l'information au niveau du bit.

Lors des jeux de tests pour des raisons de simplicité de lecture il est conseillé de ne rentrer que des valeurs entières portant sur 8 bits.

La classe Application8Bits en général :

```
class Application8Bits {
//  Integer.MAX_VALUE : 32  bit = 2147483647
//  long.MAX_VALUE : 64 bits = 9223372036854775808

  static void main(String[] args){

  static  int  BitSET(int nbr, int num)

  static  int  BitCLR(int nbr, int num)

  static int  BitCHG(int nbr, int num)

  static int DecalageD (int nbr, int n)

  static int DecalageG (int nbr, int n)

  static int BitRang (int nbr, int rang)

  static int  SetValBit (int nbr, int rang,int val)

  static int ROL (int nbr, int n)

  static int ROR (int nbr,int n)
}
```

La méthode main de la classe Application8Bits :

```
static void main(String[] args){
  int n,p,q,r,t;
  n =9;// 000...1001
  System.out.println("n=9 : n="+Integer.toBinaryString(n));
  p = BitCLR(n,3);// p=1
  System.out.println("BitCLR(n,3) ="+Integer.toBinaryString(p));
  q = BitSET(n,2);// q=13
  System.out.println("BitSET(n,2) ="+Integer.toBinaryString(q));
  r = BitCHG(n,3);// r=1
  System.out.println("BitCHG(n,3) ="+Integer.toBinaryString(r));
  t = BitCHG(n,2);// t=13
  System.out.println("BitCHG(n,2) ="+Integer.toBinaryString(t));
  System.out.println("p = "+p+", q = "+q+", r = "+r+", t = "+t);
  n =-2147483648;//1000.....00 entier minimal
  System.out.println("n=-2^31 : n="+Integer.toBinaryString(n));
  p=ROL(n,3);// 000...000100 => p=4
  System.out.println("p = "+p);
  n =-2147483648+1;//1000.....01 entier minimal+1
  System.out.println("n=-2^31+1 : n="+Integer.toBinaryString(n));
  p=ROL(n,3);// 000...0001100 => p=12
  System.out.println("p = "+p);
  n =3;//0000.....0 11
  System.out.println("n=3 : n="+Integer.toBinaryString(n));
  p=ROR(n,1);//100000...001 => p=-2147483647
  System.out.println("ROR(n,1) = "+p+"= "+Integer.toBinaryString(p));
  p=ROR(n,2);// 11000...000 => p= -1073741824
  System.out.println("ROR(n,2) = "+p+"= "+Integer.toBinaryString(p));
  p=ROR(n,3);// 011000...000 => p= +1610612736 =2^30+2^29
  System.out.println("ROR(n,3) = "+p+"= "+Integer.toBinaryString(p));
}
```

Les autres méthodes de la classe Application8Bits :

```
static  int  BitSET(int nbr, int num)
{ // positionne à 1 le bit de rang num
  int mask;
  mask =1<< num;
  return nbr | mask;
}

static  int  BitCLR(int nbr, int num)
{ // positionne à 0 le bit de rang num
  int mask;
  mask = ~ (1<< num);
  return nbr & mask;
}

static int  BitCHG(int nbr, int num)
{ // complémente le bit de rang num (0 si bit=1, 1 si bit=0)
  int mask;
  mask =1<< num;
  return nbr ^ mask;
}
```

Les autres méthodes de la classe Application8Bits (suite) :

```
static int DecalageD (int nbr, int n)
{ // décalage sans le signe de n bits vers la droite
  return nbr >>> n ;
}

static int DecalageG (int nbr, int n)
{ // décalage de 2 bits vers la gauche
  return nbr << n ;
}

static int BitRang (int nbr, int rang)
{ //renvoie le bit de rang fixé
  return (nbr >>> rang ) % 2;
}

static int  SetValBit (int nbr, int rang,int val)
{ //positionne à val le bit de rang fixé
  return    val ==0 ? BitCLR( nbr,rang):BitSET( nbr,rang) ;
}

static int ROL (int nbr, int n)
{ //décalage à gauche avec rotation
  int C;
  int N = nbr;
  for(int i=1; i<=n; i++)
  {
    C = BitRang(N,31);
    N = N <<1;
    N = SetValBit(N,0,C);
  }
  return N ;
}

static int ROR (int nbr,int n)
{ //décalage à droite avec rotation
  int C;
  int N = nbr;
  for(int i=1; i<=n; i++)
  {
    C = BitRang(N,0);
    N = N >>> 1;
    N = SetValBit(N,31,C);
  }
  return N ;
}
```

Résultats produits par la méthode main :

Nous avons utilisé `Integer.toBinaryString(n)` qui est la méthode de conversion de la classe Integer permettant d'obtenir sous forme d'une chaîne le contenu transformé en binaire d'une mémoire de type int. Cette méthode nous permet d'ausculter le contenu d'une mémoire en binaire afin de voir ce qui s'est passé lors de l'utilisation d'une des méthodes précédentes.

Le programme Java précédent produit les lignes suivantes :

n=9 : n=1001
BitCLR(n,3) =1
BitSET(n,2) =1101

```
BitCHG(n,3) =1
BitCHG(n,2) =1101
p = 1, q = 13, r = 1, t = 13
n=-2^31 : n=10000000000000000000000000000000
p = 4
n=-2^31+1 : n=10000000000000000000000000000001
p = 12
n=3 : n=11
ROR(n,1) = -2147483647= 10000000000000000000000000000001
ROR(n,2) = -1073741824= 11000000000000000000000000000000
ROR(n,3) = 1610612736= 11000000000000000000000000000000
```

Méthodes de traitement de chaînes

Objectif : Soit à implémenter sous forme de méthodes Java d'une classe, les six procédures ou fonctions spécifiées ci-dessous en Delphi.

function Length(S:string): Integer;

La fonction Length renvoie le nombre de caractères effectivement utilisés dans la chaîne ou le nombre d'éléments dans le tableau.

function Concat(s1, s2 : string): string;

Utilisez Concat afin de concaténer deux chaînes. Chaque paramètre est une expression de type chaîne. Le résultat est la concaténation des deux paramètres chaîne.

L'utilisation de l'opérateur plus (+) sur deux chaînes a le même effet que l'utilisation de la fonction Concat :

S := 'ABC' + 'DEF';

procedure Delete(**var** S: string; Index, Count:Integer);

La procédure Delete supprime une sous-chaîne de Count caractères dans la chaîne qui débute à la position S[Index]. S est une variable de type chaîne. Index et Count sont des expressions de type entier.

Si Index est plus grand que la taille de S, aucun caractère n'est supprimé. Si Count spécifie un nombre de caractères supérieur à ceux qui restent en partant de S[Index], Delete supprime tous les caractères jusqu'à la fin de la chaîne.

procedure Insert(Source: string; **var** S: string; Index: Integer);

Insert fusionne la chaîne Source dans S à la position S[index].

function Copy(S; Index, Count: Integer): string;

S est une expression du type chaîne. Index et Count sont des expressions de type entier. Copy renvoie une sous-chaîne ou un sous-tableau contenant Count caractères ou éléments en partant de S[Index].

function Pos (Substr: string; S: string): Integer;

La fonction Pos recherche une sous-chaîne, Substr, à l'intérieur d'une chaîne. Substr et S sont des expressions de type chaîne. Pos recherche Substr à l'intérieur de S et renvoie une valeur entière correspondant à l'indice du premier caractère de Substr à l'intérieur de S. Pos fait la distinction majuscules/minuscules. Si Substr est introuvable, Pos renvoie zéro.

Proposition de squelette de classes Java à implanter :

*La classe string contenant les méthodes de traitement des **String** :*

```
class string {

  static String delete(String s,int debut, int fin) {

  static String insert(String Source, String s, int index) {

  static int length(String s) {

  static String concat(String s1, String s2) {

  static String concat(String s1, String s2, String s3) {

  static String concat(String s1, String s2, String s3, String s4) {

  static String copy(String s, int index, int count) {

  static int pos(String substr, String s) {

}
```

La classe principale contenant la méthode main et utilisant les méthodes de la classe string :

```
class testExostring {
 //--> il n'y a pas de confusion possible entre string et String !

  public static void main(String[] Args) {
    String s = "abcdefghijklmn";
    System.out.println("s = "+s);
    System.out.println("length(s) = "+string.length(s));
    System.out.println("copy(s,2,6) = "+string.copy(s,2,6));
    System.out.println("delete(s,2,6) = "+string.delete(s,2,6));
    System.out.println("insert('xyz',s,2) = "+string.insert("xyz",s,2));
    System.out.println("pos('efghi',s) = "+string.pos("efghi",s));
    System.out.println("pos('efghk',s) = "+string.pos("efghk",s));
  }
}
```

Solution

Méthodes de traitement de chaînes

La classe string utilisant des méthodes de la classe **String** :

```
class string {

  static String delete(String s,int debut, int fin) {
    String t1 = s.substring(0,debut-1);
    String t2 = s.substring(fin,s.length());
    return   t1+t2;
  }

  static String insert(String Source, String s, int index) {
    String t1 = s.substring(0,index-1);
    String t2 = s.substring(index-1,s.length());
    return   t1+Source+t2;
  }

  static int length(String s) {
    return  s.length();
  }

  static String concat(String s1, String s2) {
    return   s1+s2;
  }

  static String copy(String s, int index, int count) {
    return  s.substring(index-1,index+count-2);
  }

  static int pos(String substr, String s) {
    return s.indexOf(substr)+1;
  }

}
```

La classe principale contenant la méthode main et utilisant les méthodes de la classe string :

```
class testExostring {
  //--> il n'y a pas de confusion possible entre string et String !

  public static void main(String[] Args) {
    String s = "abcdefghijklmn";
    System.out.println("s = "+s);
    System.out.println("length(s) = "+string.length(s));
    System.out.println("copy(s,2,6) = "+string.copy(s,2,6));
    System.out.println("delete(s,2,6) = "+string.delete(s,2,6));
    System.out.println("insert('xyz',s,2) = "+string.insert("xyz",s,2));
    System.out.println("pos('efghi',s) = "+string.pos("efghi",s));
    System.out.println("pos('efghk',s) = "+string.pos("efghk",s));
  }
}
```

TD String Java
phrase palindrome (première version)

Voici le squelette du programme Java à écrire :

```
/*
  phrases palindromes :
     et la marine s'en ira, malte.
     esope reste ici et se repose.
     elu par cette crapule.
*/

class palindromel {

  static String compresser(String s){
  }

  static String Inverser(String s){
  }

  public static void main(String[] x){

    System.out.print("Entrez une phrase : ");
    String phrase=Readln.unstring();
    String strMot=compresser(phrase);
    System.out.println("strMot="+strMot);
    String strInv=Inverser(strMot);
    System.out.println("strInv="+strInv);

    if(strMot.equals(strInv))
       System.out.println("palindrome");
    else System.out.println("non palindrome");
  }
}
```

Travail à effectuer :

Ecrire les méthode **compresser** et **Inverser** , il est demandé d'écrire une première version de la méthode **Inverser**.

- La première version de la méthode **Inverser** construira une chaîne locale à la méthode caractère par caractère avec une boucle **for** à un seul indice.

Solutions String Java
phrase palindrome (première version)

La méthode **compresser** :

```
static String compresser(String s){
    String strLoc="";
    for(int i=0; i<s.length(); i++){
        if(s.charAt(i) !=' ' && s.charAt(i) !=','
            && s.charAt(i) !='\'' && s.charAt(i) !='.')
            strLoc +=s.charAt(i);
    }
    return strLoc;
}
```

La méthode **compresser** élimine les caractères non recevables comme : blanc, virgule, point et apostrophe de la **String** s passée en paramètre.

Remarquons que l'instruction strLoc +=s.charAt(i) permet de concaténer les caractères recevables de la chaîne locale strLoc, par balayage de la String s depuis le caractère de rang 0 jusqu'au caractère de rang s.length()-1.

La référence de String strLoc pointe à chaque tour de la boucle **for** vers un nouvel objet créé par l'opérateur de concaténation +

La première version de la méthode **Inverser** :

```
static String Inverser(String s){
    String strLoc="";
    for(int i=0; i<s.length(); i++)
        strLoc =s.charAt(i)+strLoc;
    return strLoc;
}
```

TD String-tableau
phrase palindrome (deuxième version)

Voici le squelette du programme Java à écrire :

```
class palindrome1 {

  static String compresser(String s){
  }

  static String Inverser(String s){
  }

  public static void main(String[] x){

    System.out.print("Entrez une phrase : ");
    String phrase=Readln.unstring();
    String strMot=compresser(phrase);
    System.out.println("strMot="+strMot);
    String strInv=Inverser(strMot);
    System.out.println("strInv="+strInv);

    if(strMot.equals(strInv))
      System.out.println("palindrome");
    else System.out.println("non palindrome");
  }
}
```

Travail à effectuer :

Ecrire les méthode **compresser** et **Inverser** , il est demandé d'écrier une deuxième version de la méthode **Inverser**.

- La deuxième version de la méthode **Inverser** modifiera les positions des caractères ayant des positions symétriques dans la chaîne avec une boucle **for** à deux indices et en utilisant un tableau de **char**.

La méthode **compresser** a déjà été développée auparavant et reste la même :

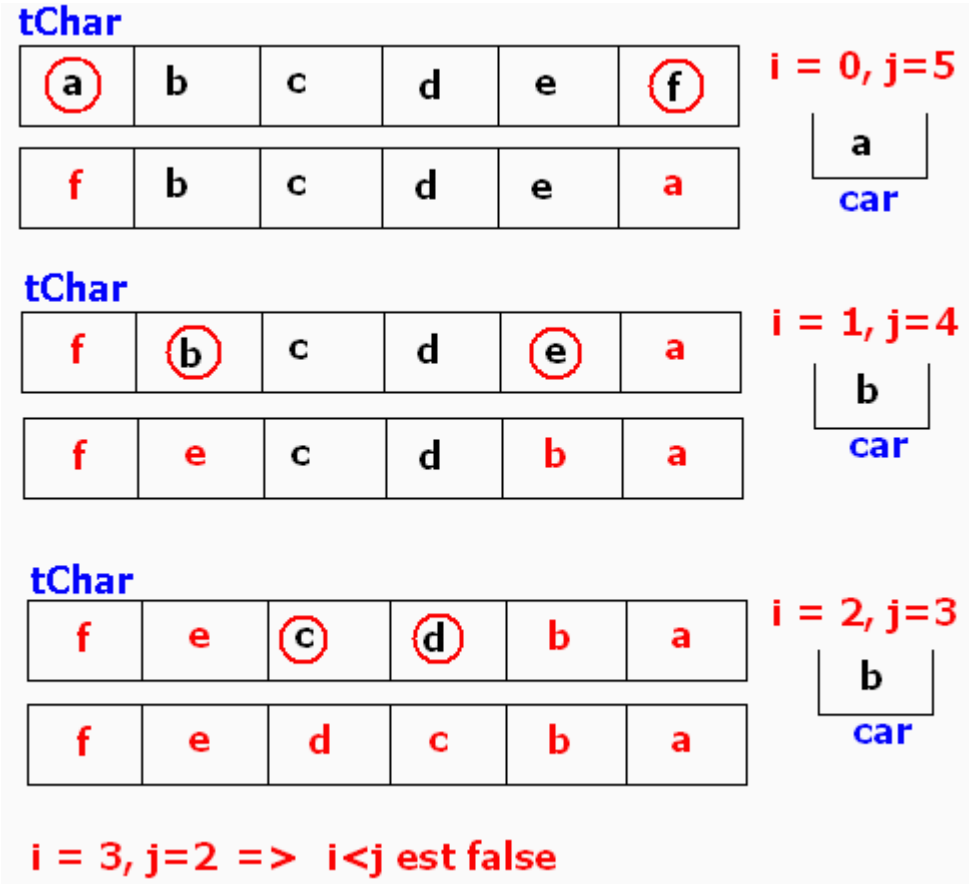
```
static String compresser(String s){
  String strLoc="";
  for(int i=0; i<s.length(); i++){
    if(s.charAt(i) !=' ' && s.charAt(i) !=','
        && s.charAt(i) !='\'' && s.charAt(i) !='.')
      strLoc +=s.charAt(i);
  }
  return strLoc;
}
```

Solution String-tableau
phrase palindrome (deuxième version)

La deuxième version de la méthode **Inverser** :

```
static String Inverser(String s){
    char [ ] tChar =s.toCharArray();
    char car ;
    for ( int i=0 , j=tChar.length-1 ; i<j; i++, j--){
        car = tChar[i];
        tChar[i ]= tChar[j];
        tChar[j] = car;
    }
    return new String(tChar);
}
```

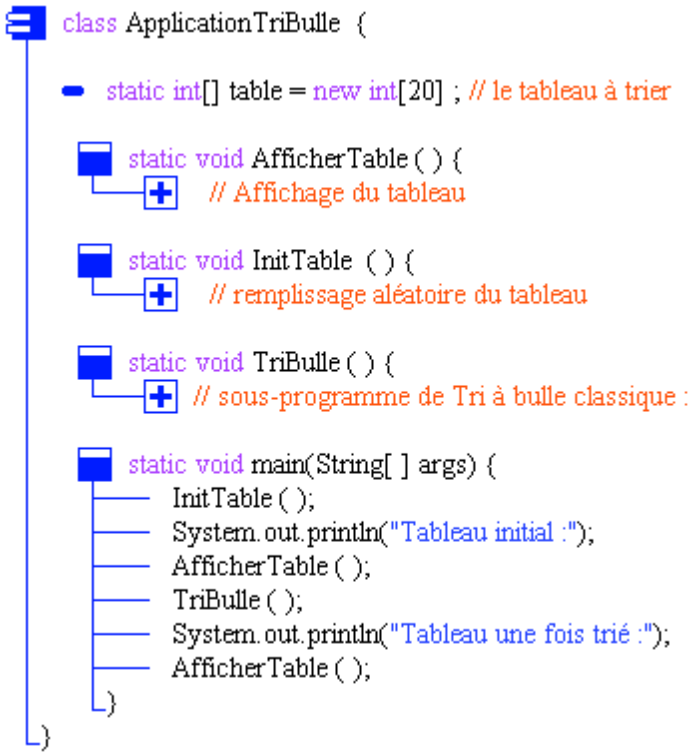
Trace d'exécution sur la chaîne s = "abcdef" :



Algorithme
Tri à bulles sur un tableau d'entiers

Objectif : Ecrire un programme Java implémentant l'algorithme du tri à bulles.

Proposition de squelette de classe Java à implanter :



```
class ApplicationTriBulle {

    static int[] table = new int[20] ; // le tableau à trier

    static void AfficherTable ( ) {
        // Affichage du tableau

    static void InitTable ( ) {
        // remplissage aléatoire du tableau

    static void TriBulle ( ) {
        // sous-programme de Tri à bulle classique :

    static void main(String[ ] args) {
        InitTable ( );
        System.out.println("Tableau initial :");
        AfficherTable ( );
        TriBulle ( );
        System.out.println("Tableau une fois trié :");
        AfficherTable ( );
    }
}
```

Spécifications de l'algorithme :

Algorithme Tri_a_Bulles
local: i , j , **n**, temp ∈ Entiers naturels
Entrée - Sortie : Tab ∈ Tableau d'Entiers naturels de **1** à **n** éléments
début
pour i **de** **n** **jusquà** 1 **faire** *// recommence une sous-suite (**a**1, **a**2, ... , **a**i)*
pour j **de** 2 **jusquà** i **faire** *// échange des couples non classés de la sous-suite*
si Tab[j-1] > Tab[j] **alors** *// **a**j-1et **a**j non ordonnés*
temp ←Tab[j-1] ;
Tab[j-1] ← Tab[j] ;
Tab[j] ← temp *//on échange les positions de **a**j-1et **a**j*
Fsi
fpour
fpour
Fin Tri_a_Bulles

Solution en Java
Tri à bulles sur un tableau d'entiers

La méthode Java implantant l'algorithme de tri à bulle :

```
static void TriBulle ( ) {
/* sous-programme de Tri à bulle classique :
   on trie les éléments du n°1 au  n°19   */
  int n = table.length-1;
  for ( int i = n; i>=1; i--)
     for ( int j = 2; j <= i; j++)
        if (table[j-1] > table[j]){
           int temp = table[j-1];
           table[j-1] = table[j];
           table[j] = temp;
        }
}
```

Une classe contenant cette méthode et la testant :

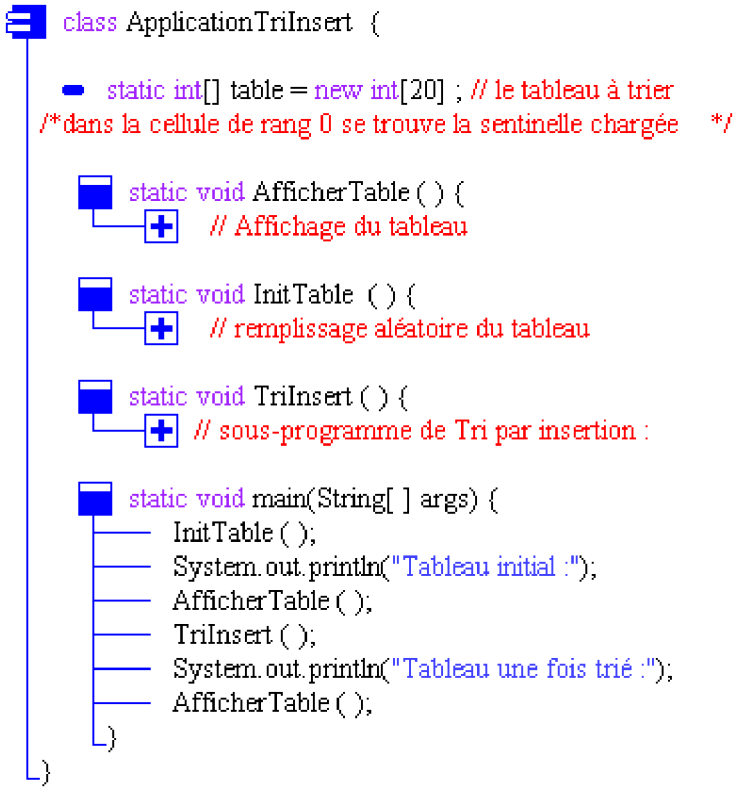
```
class ApplicationTriBulle  {
  //-- le tableau à trier : 20 éléments
  static int[] table = new int[20] ;

  static void TriBulle ( ) {
     /* sous-programme de Tri à bulle classique :

  static void AfficherTable ( ) {
    //-- Affichage du tableau
    int n = table.length-1;
    for ( int i = 1; i <= n; i++)
       System.out.print(table[i]+" , ");
    System.out.println();
  }

  static void InitTable  ( ) {
    //-- remplissage aléatoire du tableau
    int n = table.length-1;
    for ( int i = 1; i <= n; i++)
       table[i] = (int)(Math.random()*100);
  }

  public static void main(String[ ] args) {
    InitTable ( );
    System.out.println("Tableau initial :");
    AfficherTable ( );
    TriBulle ( );
    System.out.println("Tableau une fois trié :");
    AfficherTable ( );
  }
}
```

Tableau initial :
3 , 97 , 27 , 2 , 56 , 67 , 25 , 87 , 41 , 2 , 80 , 73 , 61 , 97 , 46 , 92 , 38 , 70 , 32 ,
Tableau une fois trié :
2 , 2 , 3 , 25 , 27 , 32 , 38 , 41 , 46 , 56 , 61 , 67 , 70 , 73 , 80 , 87 , 92 , 97 , 97 ,

Algorithme
Tri par insertion sur un tableau d'entiers

Objectif : Ecrire un programme Java implémentant l'algorithme du tri par insertion.

Proposition de squelette de classe Java à implanter :



Spécifications de l'algorithme :

Algorithme Tri_Insertion
local: i , j , **n**, v ∈ Entiers naturels
Entrée : Tab ∈ Tableau d'Entiers naturels de 0 à **n** éléments
Sortie : Tab ∈ Tableau d'Entiers naturels de 0 à **n** éléments (le même tableau)

{ *dans la cellule de rang 0 se trouve une sentinelle chargée d'éviter de tester dans la boucle* **tantque** .. **faire** *si l'indice j n'est pas inférieur à 1, elle aura une valeur inférieure à toute valeur possible de la liste*
}

```
début


pour i de2 jusquà n faire// la partie non encore triée (ai, ai+1, ... , an)
   v ← Tab[ i ] ;  // l'élément frontière : ai
   j ← i ;        // le rang de l'élément frontière
    Tantque Tab[ j-1 ]> v faire//on travaille sur la partie déjà triée (a1, a2, ... , ai)
          Tab[ j ]  ←Tab[ j-1 ]; // on décale l'élément
           j ← j-1;              // on passe au rang précédent
    FinTant ;
    Tab[ j ]  ← v //on recopie ai dans la place libérée
 fpour


Fin Tri_Insertion
```

On utilise une sentinelle placée dans la cellule de rang 0 du tableau, comme le type d'élément du tableau est un **int**, nous prenons comme valeur de la sentinelle une valeur négative très grande par rapport aux valeurs des éléments du tableau; par exemple le plus petit élément du type int, soit la valeur Integer.MIN_VALUE.

Solution en Java
Tri par insertion sur un tableau d'entiers

La méthode Java implantant l'algorithme de tri par insertion :

```
static void TriInsert ( ) {
// sous-programme de Tri par insertion :
   int n = table.length-1;
   for ( int i = 2; i <= n; i++) {
      int v = table[i];
      int j = i;
      while (table[ j-1 ] > v) {//on travaille sur la partie déjà triée (a1, a2, ... , ai)
         table[ j ] = table[ j-1 ]; // on décale l'élément
         j = j-1;                   // on passe au rang précédent
      } ;
      table[ j ]  = v ; //on recopie ai dans la place libérée
   }
}
```

Une classe contenant cette méthode et la testant :

```
class ApplicationTriInsert  {
   // le tableau à trier : 20 éléments
  static int[] table = new int[20] ;

  static void TriInsert ( ) {
     /* sous-programme de Tri par insertion :

  static void AfficherTable ( ) {
  //-- Affichage du tableau
     int n = table.length-1;
     for ( int i = 0; i <= n; i++)
        System.out.print(table[i]+" , ");
     System.out.println();
  }

  static void InitTable  ( ) {
  //-- remplissage aléatoire du tableau
     int n = table.length-1;
     for ( int i = 1; i <= n ; i++)
        table[i] = (int)(Math.random()*100);
     table[0] = -Integer.MAX_VALUE; //sentinelle à l'indice 0
  }

  public static void main(String[ ] args) {
     InitTable ( );
     System.out.println("Tableau initial :");
     AfficherTable ( );
     TriInsert ( );
     System.out.println("Tableau une fois trié :");
     AfficherTable ( );
  }
}
```

Algorithme

Recherche linéaire dans une table non triée

Objectif : Ecrire un programme Java effectuant une recherche séquentielle dans un tableau linéaire (une dimension) non trié

TABLEAU NON TRIE

Spécifications de l'algorithme :

- Soit **t** un tableau d'entiers de **1..n** éléments non rangés.

- On recherche le rang (la place) de l'élément **Elt** dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément **Elt** n'est pas présent dans le tableau **t**)

Version **Tantque** avec "**et alors**" (opérateur et optimisé)

```
 i ← 1 ;
Tantque (i ≤ n) et alors (t[i] ≠ Elt) faire
      i ← i+1
finTant;
si  i ≤ n  alors rang ← i
sinon rang ← -1
Fsi
```

Version **Tantque** avec "**et**" (opérateur et non optimisé)

```
 i ← 1 ;
Tantque (i < n) et (t[i] ≠ Elt) faire
      i ← i+1
finTant;
si  t[i] = Elt alors rang ← i
sinon rang ← -1
Fsi
```

Version **Tantque** avec sentinelle en fin de tableau (rajout d'une cellule)

```
t[n+1] ← Elt ; // sentinelle rajoutée
i ← 1 ;
Tantque (i ≤ n) et alors (t[i] ≠ Elt) faire
      i ←  i+1
finTant;
si  i ≤ n  alors rang ←  i
sinon rang ← -1
Fsi
```

Version **Pour** avec instruction de sortie (**Sortirsi**)

```
pour i ← 1 jusquà n faire
   Sortirsi t[i] = Elt
fpour;
si  i ≤ n  alors rang ← i
sinon rang ←  -1
Fsi
```

Traduire chacune des quatre versions sous forme d'une méthode Java.

Proposition de squelette de classe Java à implanter :

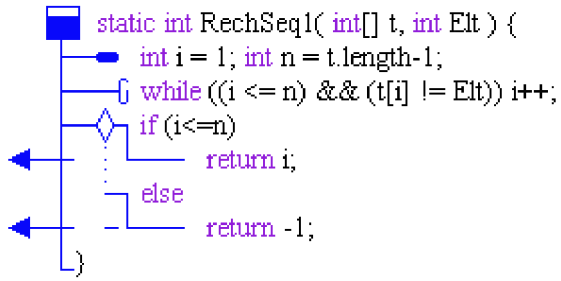
```
class ApplicationRechLin  {

      static int max = 20;
      static int[ ] table = new int[max] ; //20 cellules à examiner de 1 à 19
      static int[ ] tableSent = new int[max+1] ; // le tableau à examiner de 1 à 20


 static void AfficherTable (int[ ] t ) {
         // Affichage du tableau
        int n = t.length-1;
        for ( int i = 1; i <= n; i++)
          System.out.print(t[i]+" , ");
        System.out.println();
  }



 static void InitTable  ( ) {
         // remplissage aléatoire du tableau
        int n = table.length-1;
        for ( int i = 1; i <= n; i++) {
```

```
            table[i] = (int)(Math.random( )*100);
            tableSent[i] = table[i];
          }
    }

    static int RechSeq1( int[ ] t, int Elt ) {

    }

    static int RechSeq2( int[ ] t, int Elt ) {

    }

    static int RechSeq3( int[ ] t, int Elt ) {

    }

   static int RechSeq4( int[ ] t, int Elt ) {

   }

   public static void main(String[ ] args) {
      InitTable ( );
      System.out.println("Tableau initial :");
      AfficherTable (table );
      int x = Readln.unint(), rang;
      //appeler ici les méthodes de recherche…
      if (rang > 0)
         System.out.println("Elément "+x+" trouvé en : "+rang);
      else  System.out.println("Elément "+x+" non trouvé !");
    }
}
```

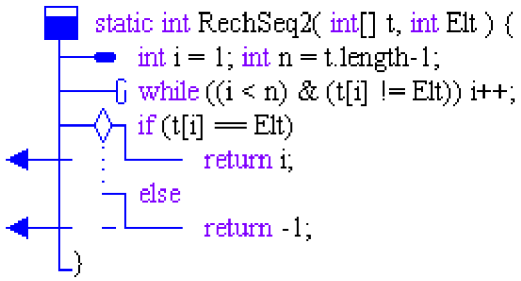
Solution en Java
Recherche linéaire dans une table non triée

Les différentes méthodes Java implantant les 4 versions d'algorithme de recherche linéaire (table non triée) :

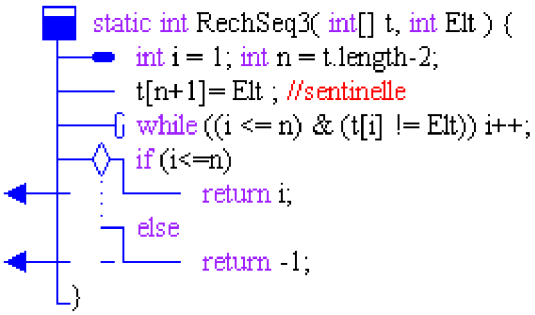
Version **Tantque** avec "**et alors**" (optimisé)



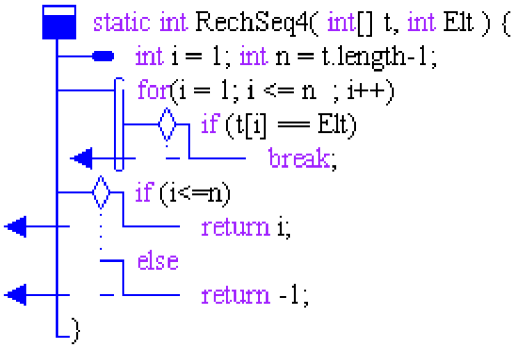
Version **Tantque** avec "**et**" (non optimisé)



Version **Tantque** avec sentinelle à la fin



Version **Pour** avec break




```
class ApplicationRechLin  {

      static int max = 20;
      static int[ ] table = new int[max] ; //20 cellules à examiner de 1 à 19
      static int[ ] tableSent = new int[max+1] ; // le tableau à examiner de 1 à 20


 static void AfficherTable (int[ ] t ) {
         // Affichage du tableau
        int n = t.length-1;
        for ( int i = 1; i <= n; i++)
          System.out.print(t[i]+" , ");
        System.out.println();
  }
```

```
  static void InitTable  ( ) {
          // remplissage aléatoire du tableau
         int n = table.length-1;
         for ( int i = 1; i <= n; i++) {
           table[i] = (int)(Math.random( )*100);
           tableSent[i] = table[i];
         }
  }

  static int RechSeq1( int[ ] t, int Elt ) { … }

  static int RechSeq2( int[ ] t, int Elt ) { … }

  static int RechSeq3( int[ ] t, int Elt ) { … }

 static int RechSeq4( int[ ] t, int Elt ) { … }

 public static void main(String[ ] args) {
    InitTable ( );
    System.out.println("Tableau initial :");
    AfficherTable (table );
    int x = Readln.unint(), rang;
    //rang = RechSeq1( table, x );
    //rang = RechSeq2( table, x );
    //rang = RechSeq3( tableSent, x );
    rang = RechSeq4( table, x );
    if (rang > 0)
      System.out.println("Elément "+x+" trouvé en : "+rang);
    else  System.out.println("Elément "+x+" non trouvé !");
  }
}
```

Algorithme

Recherche linéaire dans une table déjà triée

Objectif : Ecrire un programme Java effectuant une recherche séquentielle dans un tableau linéaire (une dimension) déjà trié.

TABLEAU DEJA TRIE

Spécifications de l'algorithme :

- Soit **t** un tableau d'entiers de **1..n** éléments rangés par ordre croissant par exemple.
- On recherche le rang (la place) de l'élément **Elt** dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément **Elt** n'est pas présent dans le tableau **t**)

On peut reprendre sans changement les algorithmes précédents travaillant sur un tableau non trié.

On peut aussi utiliser le fait que le dernier élément du tableau est **le plus grand élément** et s'en servir comme une sorte de **sentinelle**. Ci-dessous deux versions utilisant cette dernière remarque.

Version **Tantque** :

```
si t[n] < Elt alors rang ←  -1
sinon
   i ←  1 ;
   Tantque t[i] < Elt faire
      i ←  i+1;
    finTant;
    si t[i] =  Elt alors rang ←  i
    sinon rang ← -1 Fsi
Fsi
```

Version **pour** :

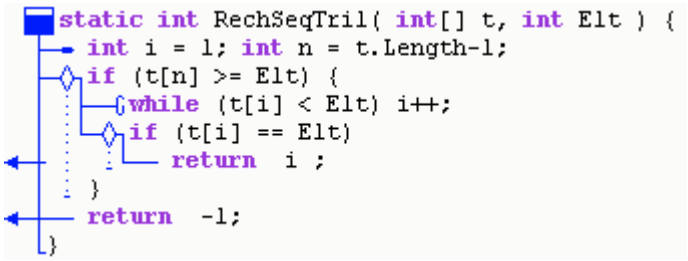
```
si t[n] < Elt alors rang ← -1
sinon
   pour i ← 1 jusquà n-1 faire
      Sortirsi t[i] ≥ Elt    // sortie de la boucle
   fpour;
   si t[i] =  Elt alors rang ←  i
   sinon rang ←  -1 Fsi
Fsi
```

Ecrire chacune des méthodes associées à ces algorithmes (prendre soin d'avoir trié le tableau auparavant par exemple par une méthode de tri), squelette de classe proposé :

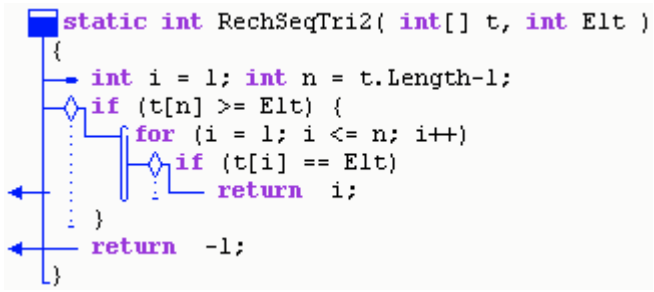
```
class ApplicationRechLinTrie  {

   static int[ ] table = new int[20] ; //20 cellules à examiner de 1 à 19


   static void AfficherTable (int[ ] t ) {
       // Affichage du tableau
      int n = t.length-1;
      for ( int i = 1; i<=n; i++)
         System.out.print(t[i]+" , ");
      System.out.println( );
   }
  static void InitTable  ( ) {
       // remplissage aléatoire du tableau
      int n = table.length-1;
      for ( int i = 1; i<=n; i++) {
         table[i] = (int)(Math.random( )*100);
      }
   }
  static void TriInsert ( ) { // sous-programme de Tri par insertion ...  }


  static int RechSeqTri1( int[] t, int Elt ) {…}


  static int RechSeqTri2( int[] t, int Elt ) {…}

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

Solution en Java
Recherche linéaire dans une table déjà triée

Les deux méthodes Java implantant les 2 versions d'algorithme de recherche linéaire (table déjà triée) :



```
static int RechSeqTri1( int[] t, int Elt ) {
  int i = 1; int n = t.Length-1;
  if (t[n] >= Elt) {
    while (t[i] < Elt) i++;
    if (t[i] == Elt)
      return  i ;
  }
  return  -1;
}
```



```
static int RechSeqTri2( int[] t, int Elt )
{
  int i = 1; int n = t.Length-1;
  if (t[n] >= Elt) {
    for (i = 1; i <= n; i++)
      if (t[i] == Elt)
        return  i;
  }
  return  -1;
}
```

La méthode main de la classe ApplicationRechLinTrie :

public static void main(String[ ] args)

```
{
        InitTable ( );
        System.out.println("Tableau initial :");
        AfficherTable (table );
        TriInsert ( );
        System.out.println("Tableau trié :");
        AfficherTable (table );
        int x = Readln.unint( ), rang;
        //rang = RechSeqTri1( table, x );
        rang = RechSeqTri2( table, x );
        if (rang > 0)
          System.out.println("Elément "+x+" trouvé en : "+rang);
        else  System.out.println("Elément "+x+" non trouvé !");

}
```

Algorithme
Liste triée de noms en Java

Objectif : Effectuer un travail de familiarisation avec la structure de liste dynamique adressable triée correspondant à la notion de tableau dynamique trié. Ce genre de structure cumule les avantages d'une structure de liste linéaire (insertion, ajout,...) et d'un tableau autorisant les accès direct par un index.

La classe concernée se dénomme Vector, elle hérite de la classe abstraite AbstractList et implémente l'interface List (public class Vector **extends** AbstractList **implements** List **)**

Nous allons utiliser un Vector pour implanter une liste triée de noms, les éléments contenus dans la liste sont des chaînes de caractères (des noms)..

Question n°1:

Codez la méthode "initialiser" qui permet de construire la liste suivante :
Liste = (voiture, terrien, eau, pied, traineau, avion, source, terre, xylophone, mer, train, marteau).

Codez la méthode "ecrire" qui permet d'afficher le contenu de la liste et qui produit l'affichage suivant :

voiture, terrien, eau, pied, traineau, avion, source, terre, xylophone, mer, train,
marteau,
Taille de la liste chaînée = 12

squelette proposé pour chaque méthode :
static void initialiser (Vector L) {....}
static void ecrire(Vector L) {....}

Remarque importante :
Une entité de classe Vector est un objet. un paramètre Java de type objet est une référence, donc nous n'avons pas le problème du passage par valeur du contenu d'un objet. En pratique cela signifie que lorsque le paramètre est un objet, il est à la fois en entrée et en sortie. Ici le Vector L est modifié par toute action interne effectuée sur lui dans les méthodes "initialiser" et "ecrire".

Question n °2:

Ecrire une méthode permettant de trier la liste des noms par odre alphabétique croissant en utilisant l'algorithme de **tri par sélection**.

On donne l'algorithme de tri par selection suivant :

Algorithme Tri_Selection
local: m, i , j , **n**, temp ∈ Entiers naturels
Entrée : Tab ∈ Tableau d'Entiers naturels de **1** à **n** éléments
Sortie : Tab ∈ Tableau d'Entiers naturels de **1** à **n** éléments

début
pour i **de** 1 **jusquà** **n-1** **faire** *// recommence une sous-suite*
m ← i ; *// i est l'indice de l'élément frontière **a**i = Tab[i]*
pour j **de** i+1 **jusquà** **n** **faire** *// (**a**i+1, **a**2, ... , **a**n)*
si Tab[j] < Tab[m] **alors** *// **a**j est le nouveau minimum partiel*
m ← j ; *// indice mémorisé*
Fsi
fpour;
temp ← Tab[m] ;
Tab[m] ← Tab[i] ;
Tab[i] ← temp *//on échange les positions de **a**i et de **a**j*
fpour
Fin Tri_Selection

squelette proposé pour la méthode :
static void triSelect (Vector L) {....}

Question n°3:

Ecrire une méthode permettant d'insérer un nouveau nom dans une liste déjà triée, selon l'algorithme proposé ci-dessous :

L : Liste de noms déjà triée,
Elt : le nom à insérer dans la liste L.
taille(L) : le nombre d'éléments de L

début
si (la liste L est vide) **ousinon** (dernierElement de la liste L ≤ Elt) **alors**
ajouter Elt en fin de liste L
sinon
pour i ← 0 **jusquà** taille(L)-1 **faire**
si Elt ≤ Element de rang i de L **alors**
insérer Elt à cette position ;
sortir
fsi
fpour

fsi
fin

squelette proposé pour la méthode :

static void inserElem (Vector L, String Elt) {....}

Contenu proposé de la méthode main avec les différents appels :

```
static void main(String[] Args) {
    Vector Liste = new Vector();
  //---> contenu de la Liste - initialisation :
    initialiser(Liste);
    ecrire(Liste);

  //---> Tri de la liste
    System.out.println("\nListe une fois triée : ");
    triSelect(Liste);
    ecrire(Liste);

  //---> Insérer un élément dans la liste triée
    String StrInserer ="trainard";
    System.out.println("\nInsertion dans la liste de : "+StrInserer);
    inserElem(Liste, StrInserer);
    ecrire(Liste);

  //---> Contenu de la Liste - boolean remove(Object x) :
    System.out.println("\nListe.remove('pied') : ");
    Liste.remove("pied");
    ecrire(Liste);
}
```

Voici ci-dessous les méthodes de la classe Vector, principalement utiles à la manipulation d'une telle liste:

Classe **Vector** :

boolean add(Object elem)	Ajoute l'élément "elem" à la fin du Vector et augmente sa taille de un.
void add(int index, Objectelem)	Ajoute l'élément "elem" à la position spécifiée par index et augmente la taille du Vector de un.
void clear( )	Efface tous les éléments présents dans le Vector et met sa taille à zéro.
Object firstElement( )	Renvoie le premier élément du Vector (l'élément de rang 0). Il faudra le transtyper selon le type d'élément du Vector.
Object get(int index)	Renvoie l'élément de rang index du Vector. Il faudra le

	transtyper selon le type d'élément du Vector.
int indexOf(Object elem)	Cherche le rang de la première occurence de l'élément "elem" dans le Vector. Renvoie une valeur comprise entre **0** et **size()-1** si "elem" est trouvé, revoie **-1** sinon.
void insertElementAt(Object elem, intindex)	Insère l'élément "elem" à la position spécifiée par index et augmente la taille du Vector de un.
boolean isEmpty( )	Teste si le Vector n'a pas d'éléments (renvoie **true**); renvoie **false** s'il contient au moins un élément.
Object lastElement( )	Renvoi le dernier élément du Vector (l'élément de rang **size()-1**). Il faudra le transtyper selon le type d'élément du Vector.
Object remove(int index)	Efface l'élément de rang index du Vector. La taille du Vector diminue de un.
boolean remove(Object elem)	Efface la première occurence de l'élément elem du Vector, la taille du Vector diminue alors de un et renvoie true. Si elem n'est pas trouvé la méthode renvoie false et le Vector n'est pas touché.
int size( )	Renvoie la taille du Vector (le nombres d'éléments contenus dans le Vector).

Solution en Java
Liste triée de noms en Java

Question n°1:

Le sous programme Java implantant la méthode "initialiser" construisant la liste :

```
static void initialiser(Vector L) {
    L.add("voiture" );
    L.add("terrien" );
    L.add("eau" );
    L.add("pied" );
    L.add("traineau" );
    L.add("avion" );
    L.add("source" );
    L.add("terre" );
    L.add("xylophone" );
    L.add("mer" );
    L.add("train" );
    L.add("marteau" );
}
```

Le sous programme Java implantant la méthode "ecrire" affichant le contenu de la liste :

```
static void ecrire(Vector L) {
    for(int i=0; i<L.size(); i++) {
        System.out.print(L.get(i)+", ");
    }
    System.out.println("\nTaille de la liste chaînée = "+L.size());
}
```

Question n°2:

La méthode "triSelect" utilisant l'algorithme de tri par sélection sur un tableau d'entiers :

```
static void TriSelect ( ) {
  // sous-programme de Tri par sélection :
    int n = table.length-1;
    for ( int i = 1; i <= n-1; i++)
    { // recommence une sous-suite
        int m = i; // i est l'indice de l'élément frontière ai = table[ i ]
        for ( int j = i+1; j <= n; j++)   // (ai+1, a2, ... , an)
            if (table[ j ] < table[ m ]) // aj est le nouveau minimum partiel
                m = j ; // indice mémorisé
     //on échange les positions de ai et de aj :
        int temp = table[ m ];
        table[ m ] = table[ i ];
        table[ i ]= temp;
    }
}
```

Nous allons construire à partir de ce modèle la méthode **static void** triSelect (Vector L)

{....}qui travaille sur un Vector, en utilisant les remarques suivantes.

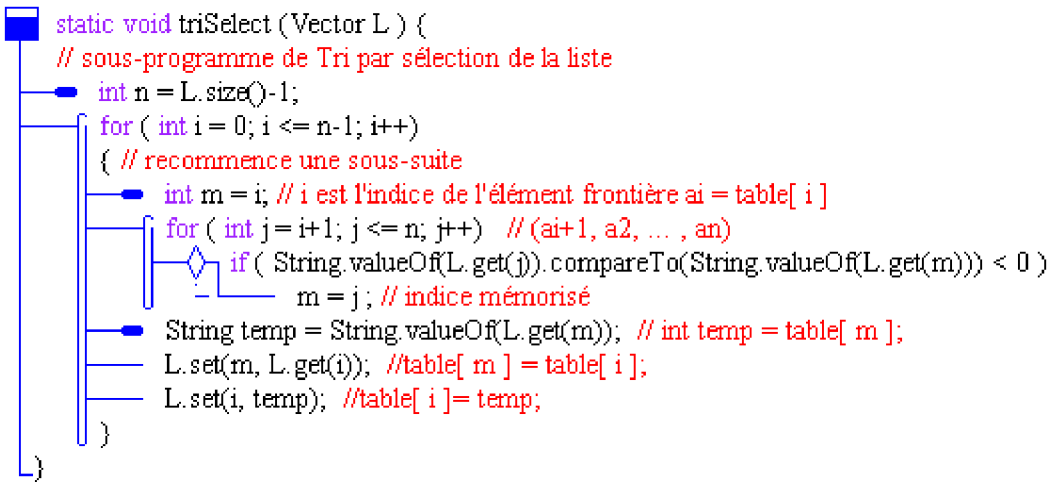
Remarques :

- Nous devons ranger des noms par ordre alphabétique croissant et non des entiers, les noms sont des String, nous devons considérer le Vector comme un tableau de String pour pouvoir le trier.
- **table[i]** de l'algorithme (accès au ième élément) est implanté en Vector par **L.get(i).**
- Comme la méthode "Object get(**int** index)" renvoie un Object nous transtypons **L.get(i)** en type String qui est un descendant d'Object, grâce à la méthode de classe valueOf de la classe String "**static** String valueOf(Object obj)", ce qui s'écrit ici : String.valueOf(**L.get(i)**).
- Les opérateurs de comparaisons <, > etc… ne prennent pas en charge le type String comme en Delphi, il est donc nécessaire de chercher dans la liste des méthodes de la classe String une méthode permettant de comparer lexicographiquement deux String.
- Pour comparer deux String (**s1** < **s2**)on trouve la méthode **compareTo** de la classe String :

  String.valueOf(L.get(j)).**compareTo**(String.valueOf(L.get(m))) < 0.

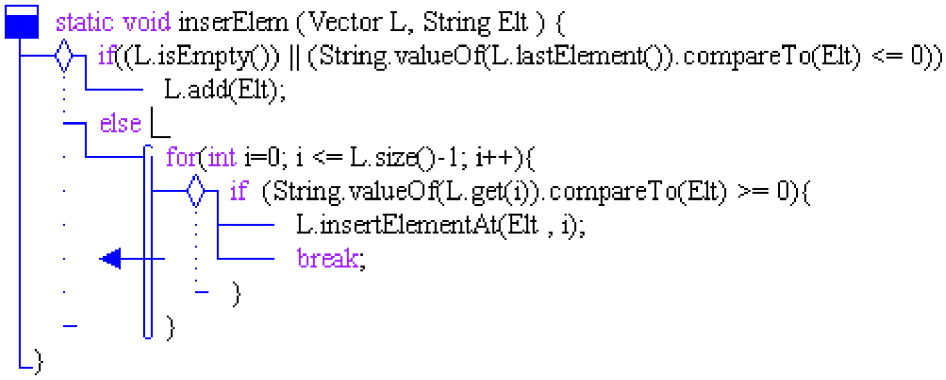
  Implantera la comparaison : **table[j]** < **table[m]**

Ce qui nous donne la méthode triSelect (Vector L) suivante :



Question n°3:

La méthode "inserElem" utilisant l'algorithme d'insertion d'un élément dans une liste triée :



```
static void inserElem (Vector L, String Elt ) {
    if((L.isEmpty()) || (String.valueOf(L.lastElement()).compareTo(Elt) <= 0))
        L.add(Elt);
    else
        for(int i=0; i <= L.size()-1; i++){
            if (String.valueOf(L.get(i)).compareTo(Elt) >= 0){
                L.insertElementAt(Elt , i);
                break;
            }
        }
}
```

Explications :

Voici les traductions utilisées pour implanter l'algorithme d'insertion :

Algorithme	Java Vector et String
la liste L est vide	L.isEmpty( )
ousinon (ou optimisé)	||
dernierElement de la liste L	L.lastElement( ) . Transtypé en String par : String.valueOf(L.lastElement( ))
dernierElement de la liste L ≤ Elt	String.valueOf(L.lastElement( )).compareTo(Elt) <= 0
ajouter Elt en fin de liste L	L.add(Elt)
taille(L)	L.size( )
Element de rang i de L	String.valueOf(L.get(i))
Elt ≤ Element de rang i de L	String.valueOf(L.get(i)).compareTo(Elt) >= 0
insérer Elt à cette position	L.insertElementAt(Elt , i)
sortir	break

Une classe complète permettant les exécutions demandées :

```
import java.util.Vector; // nécessaire à l'utilisation des Vector
 class ApplicationListeSimple
{
```

```
        static void triSelect (Vector L ) {
          // sous-programme de Tri par sélection de la liste
          int n = L.size()-1;
          for ( int i = 0; i <= n-1; i++)
          { // recommence une sous-suite
            int m = i; // i est l'indice de l'élément frontière ai = table[ i ]
            for ( int j = i+1; j <= n; j++)   // (ai+1, a2, ... , an)
              if ( String.valueOf(L.get(j)).compareTo(String.valueOf(L.get(m))) < 0 )
                 m = j ; // indice mémorisé
             String temp = String.valueOf(L.get(m));  // int temp = table[ m ];
             L.set(m, L.get(i));  //table[ m ] = table[ i ];
             L.set(i, temp);  //table[ i ]= temp;
          }
        }


    static void inserElem (Vector L, String Elt ) {
      if((L.isEmpty()) || (String.valueOf(L.lastElement()).compareTo(Elt) <= 0))
        L.add(Elt);
      else
        for(int i=0; i <= L.size()-1; i++){
          if  (String.valueOf(L.get(i)).compareTo(Elt) >= 0){
            L.insertElementAt(Elt , i);
            break;
          }
        }
    }


    static void ecrire(Vector L) {
      for(int i=0; i<L.size(); i++) {
        System.out.print(L.get(i)+", ");
      }
      System.out.println("\nTaille de la liste chaînée = "+L.size());
    }


    static void initialiser(Vector L) {
      L.add("voiture" );
      L.add("terrien" );
      L.add("eau" );
      L.add("pied" );
      L.add("traineau" );
      L.add("avion" );
      L.add("source" );
      L.add("terre" );
      L.add("xylophone" );
      L.add("mer" );
      L.add("train" );
      L.add("marteau" );
    }
```

```
    public static void main(String[] Args) {
      Vector Liste = new Vector( ); //création obligatoire d'un objet de classe Vector
     //---> contenu de la Liste - initialisation :
      initialiser(Liste);
      ecrire(Liste);

          //---> Tri de la liste :
           System.out.println("\nListe une fois triée : ");
           triSelect(Liste);
           ecrire(Liste);

          //---> Insérer un élément dans la liste triée :
           String StrInserer ="trainard";
           System.out.println("\nInsertion dans la liste de : "+StrInserer);
           inserElem(Liste, StrInserer);
           ecrire(Liste);

          //---> Contenu de la Liste - boolean remove(Object x) :
           System.out.println("\nListe.remove('pied') : ");
           Liste.remove("pied");
           ecrire(Liste);
        }

}
```

Exécution de cette classe :

voiture, terrien, eau, pied, traineau, avion, source, terre, xylophone, mer, train, marteau,
Taille de la liste chaînée = 12

Liste une fois triée :
avion, eau, marteau, mer, pied, source, terre, terrien, train, traineau, voiture, xylophone,
Taille de la liste chaînée = 12

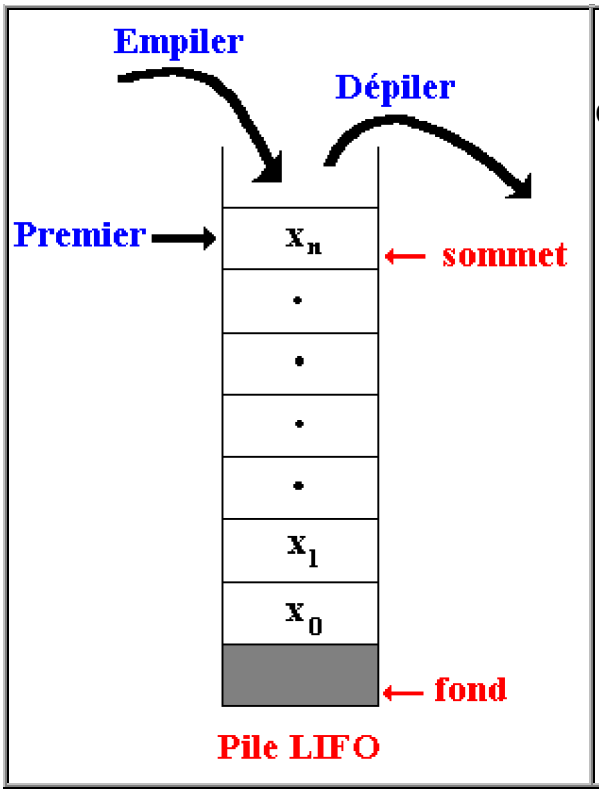
Insertion dans la liste de : trainard
avion, eau, marteau, mer, pied, source, terre, terrien, train, trainard, traineau, voiture, xylophone,
Taille de la liste chaînée = 13

Liste.remove('pied') :
avion, eau, marteau, mer, source, terre, terrien, train, trainard, traineau, voiture, xylophone,
Taille de la liste chaînée = 12

Algorithme
Structure de donnée de pile LIFO

Objectif : Nous implantons en Java une structure de pile LIFO (Last In First Out) fondée sur l'utilisation d'un objet de classse LinkedList. Nous construisons une pile LIFO de chaînes de caractères.

Rappel des spécifications d'une pile LIFO :



Opérateurs de base sur une pile LIFO :

sommet pointe vers l'élément en haut de pile, fond sert de sentinelle à la pile si nécessaire.

Dépiler ([X0, X1,..., Xn, Xn+1]) -->
Pile = [X0, X1,..., Xn] , Xn+1

Empiler([X0, X1,..., Xn] , Xn+1) -->
Pile = [X0, X1,..., Xn, Xn+1]

Premier([X0, X1,..., Xn]) = Xn

EstVide([X0, X1,..., Xn]) = **false**
EstVide([ ]) = **true** *(sommet = fond)*

Notre pile LIFO doit contenir des noms (chaînes de caractères donc utilisation des String).

La classe LinkedList est une structure dynamique (non synchronizée) qui ressemble à la classe Vector, mais qui est bien adaptée à implanter les piles et les files car elle contient des références de type Object et les String héritent des Object.

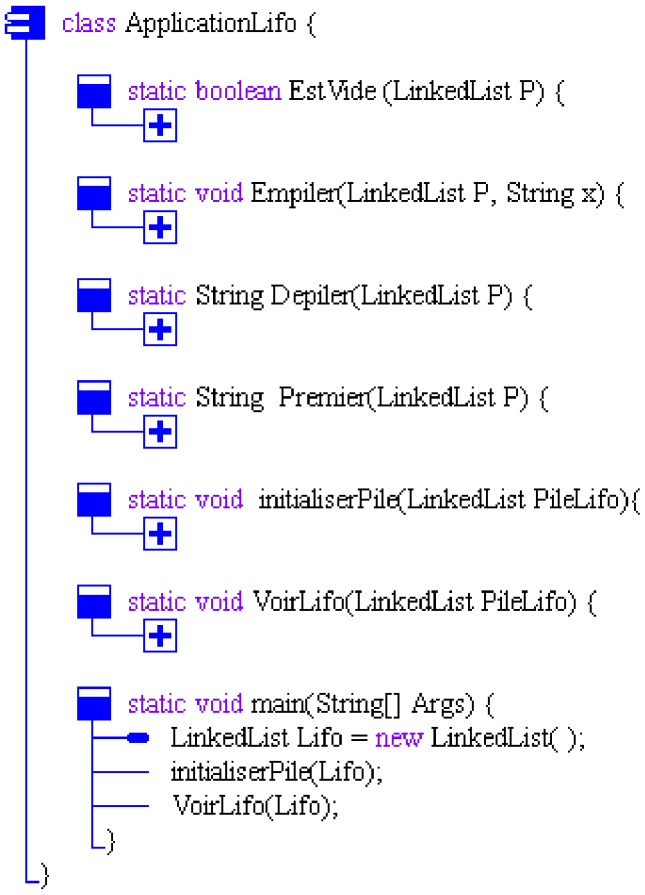
Proposition de squelette de classe Java algorithmique :

Nous utilisons un objet de classe **LinkedList** pour représenter une **pile LIFO**, elle sera passée comme paramètre dans les méthodes qui travaillent sur cet objet :

static boolean EstVide (LinkedList P)	tester si la pile P est vide

static void Empiler(LinkedList P, String x)	Empiler dans la pile P le nom x.
static String Depiler(LinkedList P)	Dépiler la pile P.
static String Premier(LinkedList P)	Renvoyer l'élément au sommet de la pile P.
static void initialiserPile(LinkedList P)	Remplir la pile P avec des noms.
static void VoirLifo(LinkedList P)	Afficher séquentiellement le contenu de P.

Complétez la classe ci-dessous et ses méthodes :



Voici ci-dessous les méthodes principalement utiles à la manipulation d'une telle liste:

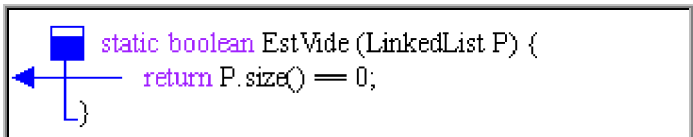
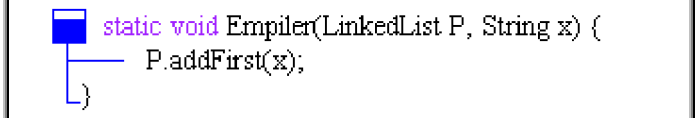
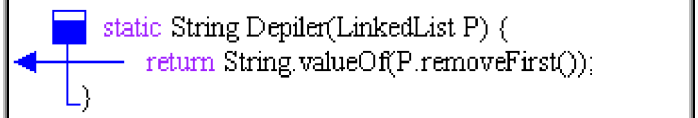
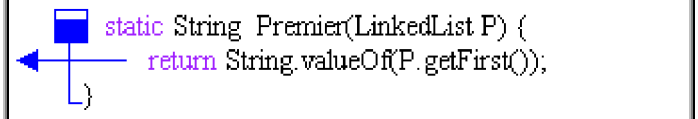
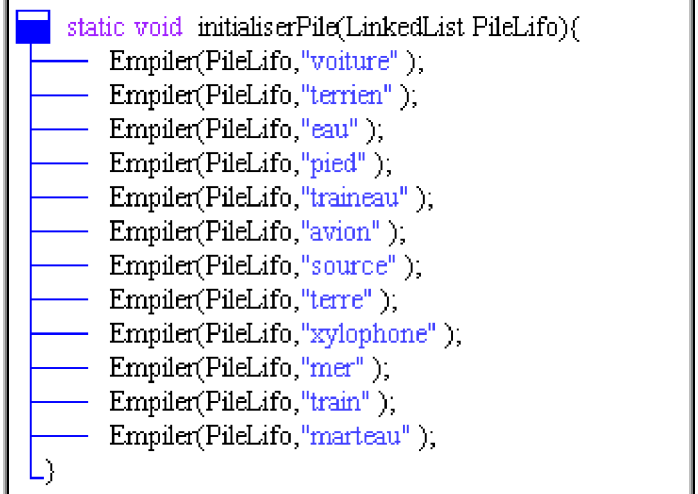
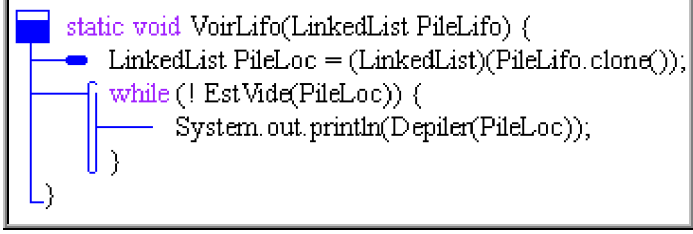
Classe **LinkedList** :

boolean add(Object elem)	Ajoute l'élément "elem" à la fin de la LinkedList et augmente sa taille de un.
void add(int index, Object elem)	Ajoute l'élément "elem" à la position spécifiée par index et augmente la taille de la LinkedList de un.
void clear( )	Efface tous les éléments présents dans la LinkedList et met sa taille à zéro.

Object getFirst()	Renvoie le premier élément de la LinkedList (l'élément en tête de liste, rang=0). Il faudra le transtyper selon le type d'élément de la LinkedList.
Object getLast()	Renvoie le dernier élément de la LinkedList (l'élément en fin de liste, rang=**size()-1**). Il faudra le transtyper selon le type d'élément de la LinkedList.
Object get(int index)	Renvoie l'élément de rang index de la LinkedList. Il faudra le transtyper selon le type d'élément de la LinkedList.
int indexOf(Object elem)	Cherche le rang de la première occurence de l'élément "elem" dans le Vector. Renvoie une valeur comprise entre **0** et **size()-1** si "elem" est trouvé, revoie **-1** sinon.
void addFirst(Object elem)	Insère l'élément "elem" en tête de la LinkedList (rang=0).
void addLast(Object elem)	Ajoute l'élément "elem" à la fin de la LinkedList et augmente sa taille de un.
boolean isEmpty( )	Teste si la LinkedList n'a pas d'éléments (renvoie **true**); renvoie **false** si elle contient au moins un élément.
Object remove(int index)	Efface l'élément de rang index de la LinkedList. La taille de la LinkedList diminue de un.
boolean remove(Object elem)	Efface la première occurence de l'élément elem de la LinkedList, la taille de la LinkedList diminue alors de un et renvoie true. Si elem n'est pas trouvé la méthode renvoie false et la LinkedList n'est pas touché.
Object removeFirst()	Efface et renvoie le premier élément (rang=0) de la LinkedList. Il faudra le transtyper selon le type d'élément de la LinkedList.
Object removeLast()	Efface et renvoie le dernier (rang=**size()-1**) élément de la LinkedList. Il faudra le transtyper selon le type d'élément de la LinkedList.
int size( )	Renvoie la taille de la LinkedList (le nombres d'éléments contenus dans la LinkedList).

Solution en Java
Structure de donnée de pile LIFO

Les méthodes s'appliquant à la pile LIFO :

	tester si la pile P est vide
	Empiler dans la pile P le nom x.
	Dépiler la pile P.
	Renvoyer l'élément au sommet de la pile P.
	Remplir la pile PileLifo avec des noms.
	Afficher séquentiellement le contenu de PileLifo.

Une classe complète permettant l'exécution des méthodes précédentes :

```
import java.util.LinkedList;

    class ApplicationLifo {

      static boolean EstVide (LinkedList P) {
        return P.size() == 0;
      }


      static void Empiler(LinkedList P, String x) {
        P.addFirst(x);
      }


      static String Depiler(LinkedList P) {
        return String.valueOf(P.removeFirst());
      }


      static String  Premier(LinkedList P) {
        return String.valueOf(P.getFirst());
      }


      static void  initialiserPile(LinkedList PileLifo){
        Empiler(PileLifo,"voiture" );
        Empiler(PileLifo,"terrien" );
        Empiler(PileLifo,"eau" );
        Empiler(PileLifo,"pied" );
        Empiler(PileLifo,"traineau" );
        Empiler(PileLifo,"avion" );
        Empiler(PileLifo,"source" );
        Empiler(PileLifo,"terre" );
        Empiler(PileLifo,"xylophone" );
        Empiler(PileLifo,"mer" );
        Empiler(PileLifo,"train" );
        Empiler(PileLifo,"marteau" );
      }


      static void VoirLifo(LinkedList PileLifo) {
        LinkedList PileLoc = (LinkedList)(PileLifo.clone());
        while (! EstVide(PileLoc)) {
          System.out.println(Depiler(PileLoc));
        }
      }
```

```
    public static void main(String[ ] Args) {
      LinkedList Lifo = new LinkedList( );
      initialiserPile(Lifo);
      VoirLifo(Lifo);
    }
}
```

Exercice

lire et écrire un enregistrement dans un fichier texte

Objectif : Nous implantons en Java une classe d'écriture dans un fichier texte d'informations sur un client et de lecture du fichier pour rétablir les informations initiales.

Chaque client est identifié à l'aide de quatre informations :

Numéro de client

Nom du client

Prénom du client

Adresse du client

Nous rangeons ces quatre informations dans le même enregistrement-client. L'enregistrement est implanté sous forme d'une ligne de texte contenant les informations relatives à un client, chaque information est séparée de la suivante par le caractère de séparation # .

Par exemple, les informations client suivantes :

Numéro de client = `12598`

Nom du client = `Dupont`

Prénom du client = `Pierre`

Adresse du client = `2, rue des moulins 37897 Thiers`

se touvent rangées dans un enregistrement constitué de quatre zones, sous la forme de la ligne de texte suivante :

```
12598#Dupont#Pierre#2, rue des moulins 37897 Thiers
```

Le fichier client se nommera "ficheclient.txt", vous écrirez les méthodes suivantes :

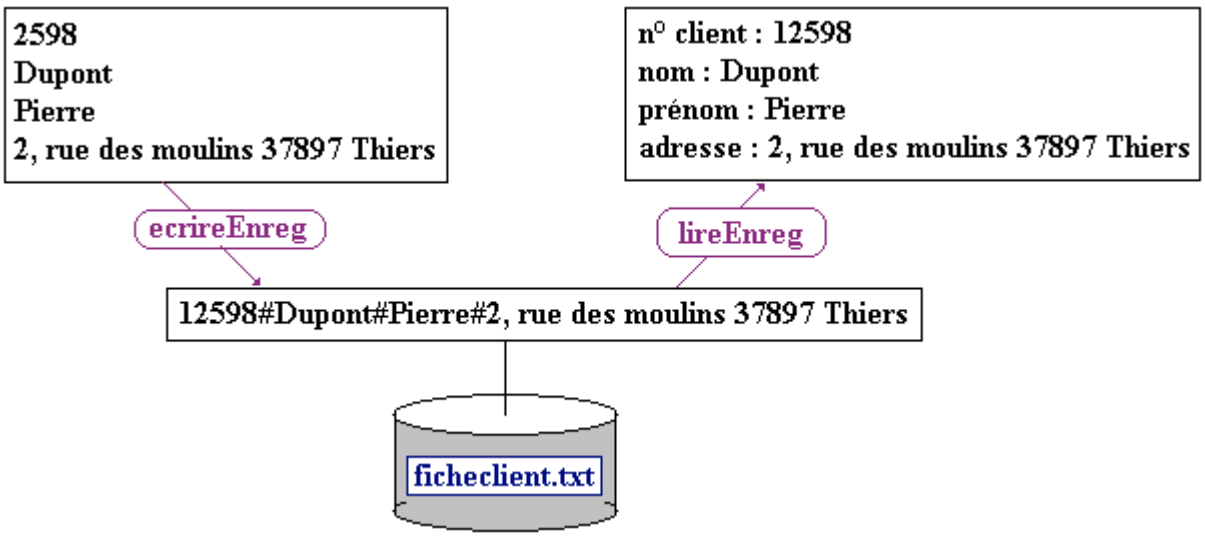
Signature de la méthode	Fonctionnement de la méthode
public static void ecrireEnreg (**String** nomFichier)	Ecrit dans le fichier client dont le nom est passé en paramètre, les informations d'un seul client sous forme d'un enregistrement (cf.ci-haut).
public static void lireEnreg (**String** nomFichier)	Lit dans le fichier client client dont le nom est passé en paramètre, un enregistrement et affiche sur la console les informations du client.
public static String[] extraitIdentite (**String**	Renvoie dans un tableau de String les 4 informations (n°, nom, prénom, adresse)

ligne)	contenues dans l'enregistrement passé en paramètre. Appelée par la méthode lireEnreg.
public static void Afficheinfo (**String**[ ] infos)	Affiche sur la console les informations du client contenues dans le tableau de String passé en paramètre. Appelée par la méthode lireEnreg.

Squelette java proposé pour la classe :

```
import java.io.*;

class AppliFichierTexte {
  private static final String separ="#";
  private static final String[] libelle=
          {"n° client : ","nom : ","prénom : ","adresse : "};

  public static void ecrireEnreg(String nomFichier) {
  }

  public static String[] extraitIdentite(String ligne){
  }

  public static void Afficheinfo(String[] infos){
  }

  public static void lireEnreg(String nomFichier) {
  }

  public static void main(String[] x){
    ecrireEnreg("ficheclient.txt");//local
    lireEnreg("ficheclient.txt");
  }
}
```

Modèle des actions effectuées par les méthodes de la classe :



Exercice

Copier un fichier texte dans un autre fichier texte

Objectif : Nous implantons en Java une classe de recopie de tout le contenu d'un fichier texte dans un nouveau fichier texte clone du premier.

Le fichier source se nommera "fiche.txt", le fichier de destination clone se dénommera "copyfiche.txt", vous écrirez les 2 méthodes suivantes :

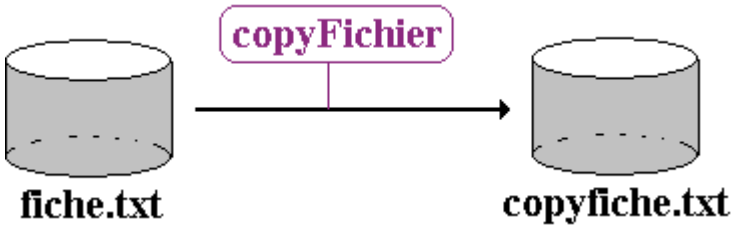
Signature de la méthode	Fonctionnement de la méthode
public static void copyFichier (**String** FichierSource, **String** FichierDest)	Copie le contenu du FichierSource dans le FichierDest.
public static void lireFichier (**String** nomFichier)	Lit tout le contenu d'un fichier client dont le nom est passé en paramètre, et affiche sur la console les informations de tout le fichier.

Squelette java proposé pour la classe :

```
import java.io.*;

class AppliCopyFichierTexte {

    public static void copyFichier(String FichierSource, String FichierDest) {

    public static void lireFichier(String nomFichier) {

    public static void main(String[] x){

}
```

Méthode main de la classe et actions :

```
public static void main(String[] x){
    System.out.println("Fichier initial : ");
    lireFichier("fiche.txt");
    copyFichier("fiche.txt","copyfiche.txt");
    System.out.println("Fichier copié : ");
    lireFichier("copyfiche.txt");
}
```



solution Java

lire et écrire un enregistrement dans un fichier texte

La classe **AppliFichierTexte** et ses membres :

```
import java.io.*;

class AppliFichierTexte {
  private static final String separ="#";
  private static final String[] libelle=
          {"n° client : ","nom : ","prénom : ","adresse : "};
   ******
  public static void main(String[] x){
    ecrireEnreg("ficheclient.txt");//local
    lireEnreg("ficheclient.txt");
  }
}

  public static void ecrireEnreg(String nomFichier) {
    try {
      FileWriter fluxwrite = new FileWriter(nomFichier);
      BufferedWriter out = new BufferedWriter(fluxwrite);
      out.write(12598+separ);
      out.write("Dupont"+separ);
      out.write("Pierre"+separ);
      out.write("2, rue des moulins 37897 Thiers");
      out.newLine( ); //écrit le eoln
      out.close( ); //sinon le fichier créé sur le disque est vide !!
    }
      catch (IOException err) {
        System.out.println( "Erreur : " + err );
      }
  }

  public static String[] extraitIdentite(String ligne){
    return ligne.split(separ);
  }

public static void Afficheinfo(String[] infos){
  for(int i=0; i<infos.length; i++)
    System.out.println(libelle[i]+infos[i]);
}

public static void lireEnreg(String nomFichier) {
  try {
    FileReader fluxread = new FileReader(nomFichier);
    BufferedReader in = new BufferedReader(fluxread);
    String Ligne;
    while( (Ligne = in.readLine()) != null) { //not eof type String (-1 autres cas)
      System.out.println("Enregistrement: "+Ligne);
      String[] info = extraitIdentite(Ligne);
      Afficheinfo(info);
    }
    in.close( );
  }
    catch (IOException err) {
      System.out.println( "Erreur : " + err );
    }
}
```

solution Java

Copier un fichier texte dans un autre fichier texte

La classe **AppliCopyFichierTexte** et ses membres :

```
import java.io.*;

class AppliCopyFichierTexte {

    public static void copyFichier(String FichierSource, String FichierDest) {
    }

    public static void lireFichier(String nomFichier) {
    }

    public static void main(String[] x){
        System.out.println("Fichier initial : ");
        lireFichier("fiche.txt");
        copyFichier("fiche.txt","copyfiche.txt");
        System.out.println("Fichier copié : ");
        lireFichier("copyfiche.txt");
    }
}
```

```
public static void copyFichier(String FichierSource, String FichierDest) {
    String Ligne;
    try {
        FileWriter fluxwrite = new FileWriter(FichierDest);
        BufferedWriter out = new BufferedWriter(fluxwrite);
        FileReader fluxread = new FileReader(FichierSource);
        BufferedReader in = new BufferedReader(fluxread);
        while( (Ligne = in.readLine()) != null) {
            out.write(Ligne);
            out.newLine();
        }
        out.close( ); //sinon le fichier créé sur le disque est vide !!
        in.close( );
    }
      catch (IOException err) {
        System.out.println( "Erreur : " + err );
      }
}
```

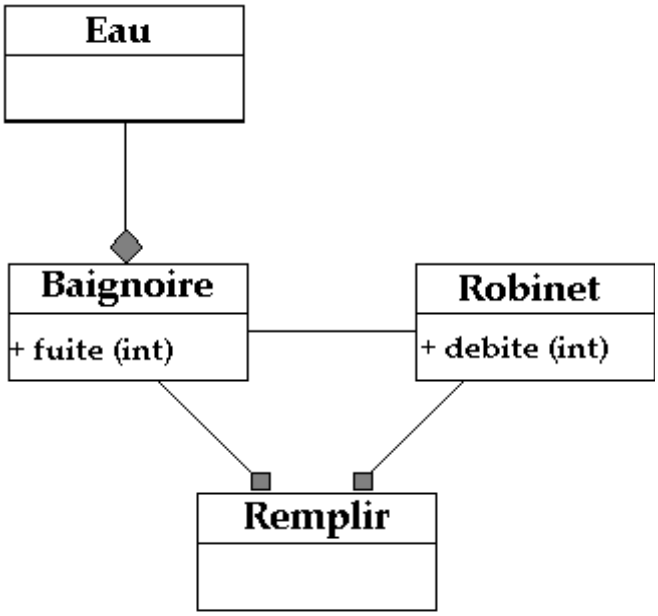
```
public static void lireFichier(String nomFichier) {
    try {
        FileReader fluxread = new FileReader(nomFichier);
        BufferedReader in = new BufferedReader(fluxread);
        String Ligne;
       //nul=eof dans le type String (-1 autres cas)
        while( (Ligne = in.readLine()) != null) {
            System.out.println(Ligne);
        }
    }
      catch (IOException err) {
        System.out.println( "Erreur : " + err );
      }
}
```

Exercice entièrement traité sur les threads

Enoncé et classes

Nous vous proposons de programmer une simulation du problème qui a tant mis à contribution les cerveaux des petits écoliers d'antan : *le problème du robinet qui remplit d'eau une baignoire qui fuit.* Nous allons écrire un programme qui simulera le remplissage de la baignoire par un robinet dont le débit est connu et paramétrable. La baignoire a une fuite dont le débit lui aussi est connu et paramétrable. Dès que la baignoire est entièrement vide l'on colmate la fuite et tout rentre dans l'ordre. On arrête le programme dès que la baignoire est pleine que la fuite soit colmatée ou non.

Nous choisissons le modèle objet pour représenter notre problème :



- Une classe **Eau** qui contient un champ static indiquant le volume d'eau actuel de l'objet auquel il appartient.
- Une classe **Baignoire** possédant un contenu (en litres d'eau) et une fuite qui diminue le volume d'eau du contenu de la baignoire.
- Une classe **Robinet** qui débite (augmente) le volume d'eau du contenu de la baignoire d'une quantité fixée.

Une première solution sans les threads

```
class Eau{
  static int volume;

  public Eau(int val){
    volume = val;
  }

}

class Robinet{
  public void debite(int quantite){
  }

}

class Baignoire {
  public static final int maximum=1000;
  public static Eau contenu = new Eau(0);

  public void fuite(int quantite ){
  }

}
```

Une classe **Remplir** qui permet le démarrage des actions : mise en place de la baignoire, du robinet, ouverture du robinet et fuite de la baignoire :

```
class Remplir {
  public static void main(String[] x){
    Baignoire UneBaignoire = new Baignoire();
    Robinet UnRobinet = new Robinet();
    UnRobinet.debite(50);
    UneBaignoire.fuite(20);
  }
}
```

Nous programmons les méthodes debite(**int** quantite) et fuite(**int** quantite) de telle sorte qu'elles afichent chacune l'état du contenu de la baignoire après que l'apport ou la diminution d'eau a eu lieu. Nous simulerons et afficherons pour chacune des deux méthodes 100 actions de base (100 diminutions pour la méthode fuite et 100 augmentations pour la méthode debite).

```
class Eau{
  static int volume;

  public Eau(int val){
    volume = val;
  }

}

class Robinet{
  public void debite(int quantite){
    for(int i=1; i<100; i++)
      if(Baignoire.contenu.volume < Baignoire.maximum){
        Baignoire.contenu.volume +=quantite;
        System.out.println("Contenu de la baignoire = "
                           +Baignoire.contenu.volume);
      }
      else {
        System.out.println("Baignoire enfin pleine !");
        break;
      }
  }

}
```

```
class Baignoire {
  public static final int maximum=1000;
  public static Eau contenu = new Eau(0);

  public void fuite(int quantite ){
    for(int i=1; i<100; i++)
      if(Baignoire.contenu.volume <= 0){
        System.out.println("Baignoire vide, on colmate la fuite !");
        break;
      }
      else if(Baignoire.contenu.volume >= Baignoire.maximum)
        break;
      else {
        Baignoire.contenu.volume -=quantite;
        System.out.println("Contenu de la baignoire = "
                           +Baignoire.contenu.volume);
      }
  }

}
```

```
class Remplir {
  public static void main(String[] x){
    Baignoire UneBaignoire = new Baignoire();
    Robinet UnRobinet = new Robinet();
    UnRobinet.debite(50);
    UneBaignoire.fuite(20);
  }
}
```

Résultats d'exécution :

```
Contenu de la baignoire = 50
Contenu de la baignoire = 100
Contenu de la baignoire = 150
Contenu de la baignoire = 200
Contenu de la baignoire = 250
Contenu de la baignoire = 300
Contenu de la baignoire = 350
Contenu de la baignoire = 400
Contenu de la baignoire = 450
Contenu de la baignoire = 500
Contenu de la baignoire = 550
Contenu de la baignoire = 600
Contenu de la baignoire = 650
Contenu de la baignoire = 700
Contenu de la baignoire = 750
Contenu de la baignoire = 800
Contenu de la baignoire = 850
Contenu de la baignoire = 900
Contenu de la baignoire = 950
Contenu de la baignoire = 1000
Baignoire enfin pleine !

---- operation complete.
```

Que s'est-il passé ?

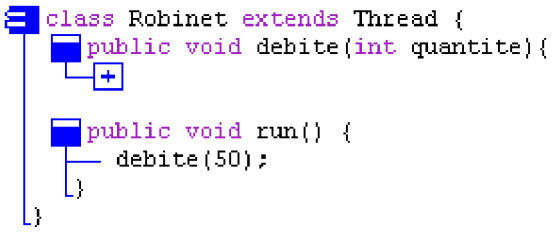
La programmation séquentielle du problème n'a pas permis d'exécuter l'action de fuite de la baignoire puisque nous avons arrêté le processus dès que la baignoire était pleine. En outre nous n'avons pas pu simuler le remplissage et le vidage "simultanés" de la baignoire.

Nous allons utiliser deux threads (secondaires) pour rendre la simulation plus réaliste, en essayant de faire les actions de débit-augmentation et fuite-diminution en parallèle.

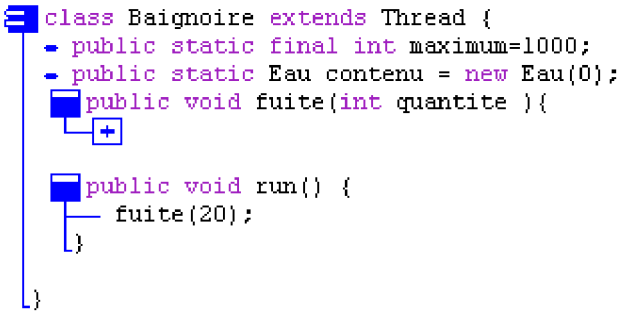
Deuxième solution avec des threads

Les objets qui produisent les variations du volume d'eau sont le robinet et la baignoire, ce sont eux qui doivent être "multi-threadés".

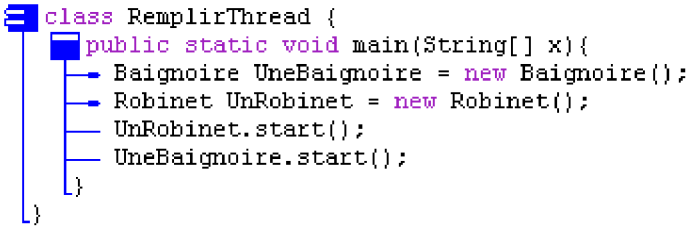
Reprenons pour cela la classe Robinet en la dérivant de la classe Thread, et en redéfinissant la méthode run( ) qui doit contenir le code de débit à exécuter en "parallèle" (le corps de la méthode **debite** n'a pas changé) :



De même en dérivant la classe Baignoire de la classe Thread, et en redéfinissant la méthode run( ) avec le code de fuite à exécuter en "parallèle" (le corps de la méthode **fuite** n'a pas changé) :



Enfin la classe RemplirThread qui permet le démarrage des actions : mise en place de la baignoire, du robinet, puis lancement en parallèle de l'ouverture du robinet et de la fuite de la baignoire :



Résultats d'exécution obtenus :

Remplissage, contenu de la baignoire = 50 Remplissage, contenu de la baignoire = 100 Remplissage, contenu de la baignoire = 150 Remplissage, contenu de la baignoire = 200 Remplissage, contenu de la baignoire = 250	Remplissage, contenu de la baignoire = 900 Fuite, contenu de la baignoire = 880 Remplissage, contenu de la baignoire = 930 Fuite, contenu de la baignoire = 910 Fuite, contenu de la baignoire = 890

```
Remplissage, contenu de la baignoire = 300
Remplissage, contenu de la baignoire = 350
Remplissage, contenu de la baignoire = 400
Remplissage, contenu de la baignoire = 450
Remplissage, contenu de la baignoire = 500
Remplissage, contenu de la baignoire = 550
Remplissage, contenu de la baignoire = 600
Fuite, contenu de la baignoire = 580
Remplissage, contenu de la baignoire = 630
Fuite, contenu de la baignoire = 610
Remplissage, contenu de la baignoire = 660
Fuite, contenu de la baignoire = 640
Remplissage, contenu de la baignoire = 690
Fuite, contenu de la baignoire = 670
Remplissage, contenu de la baignoire = 720
Remplissage, contenu de la baignoire = 770
Remplissage, contenu de la baignoire = 820
Remplissage, contenu de la baignoire = 870
Fuite, contenu de la baignoire = 850
```

```
Fuite, contenu de la baignoire = 870
Fuite, contenu de la baignoire = 850
Fuite, contenu de la baignoire = 830
Fuite, contenu de la baignoire = 810
Fuite, contenu de la baignoire = 790
Fuite, contenu de la baignoire = 770
Fuite, contenu de la baignoire = 750
Fuite, contenu de la baignoire = 730
Fuite, contenu de la baignoire = 710
Fuite, contenu de la baignoire = 690
Remplissage, contenu de la baignoire = 740
Fuite, contenu de la baignoire = 720
Remplissage, contenu de la baignoire = 770
Remplissage, contenu de la baignoire = 820
Remplissage, contenu de la baignoire = 870
Remplissage, contenu de la baignoire = 920
Remplissage, contenu de la baignoire = 970
Remplissage, contenu de la baignoire = 1020
Baignoire enfin pleine !
```

---- operation complete.

Nous voyons que les deux threads s'exécutent cycliquement (mais pas d'une manière égale) selon un ordre non déterministe sur lequel nous n'avons pas de prise mais qui dépend de la java machine et du système d'exploitation, ce qui donnera des résultats différents à chaque nouvelle exécution. Le paragraphe suivant montre un exemple ou nous pouvons contraindre des threads de "dialoguer" pour laisser la place l'un à l'autre

variation sur les threads

Lorsque la solution adoptée est l'héritage à partir de la classe Thread, vous pouvez agir sur l'ordonnancement d'exécution des threads présents. Dans notre exemple utilisons deux méthodes de cette classe Thread :

- **void** setPriority (**int** newPriority)
- **static void** yield ( )

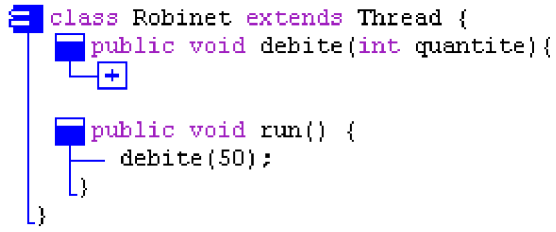
Privilégions le thread Robinet grâce à la méthode setPriority :

La classe Thread possède 3 champs **static** permettant d'attribuer 3 valeurs de priorités différentes, de la plus haute à la plus basse, à un thread indépendamment de l'échelle réelle du système d'exploitation sur lequelle travaille la Java Machine :

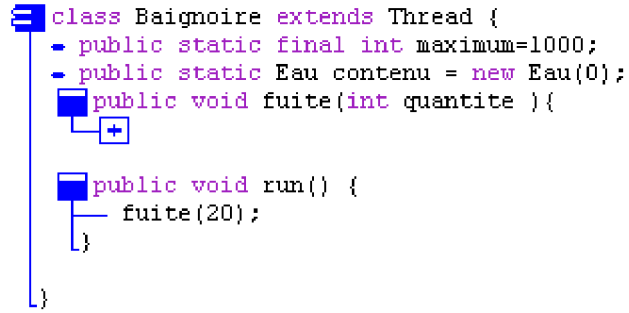
static int MAX_PRIORITY	La priorité **maximum** que peut avoir un thread.
static int MIN_PRIORITY	La priorité **minimum** que peut avoir un thread.
static int NORM_PRIORITY	La priorité **par défaut** attribuée à un thread.

La méthode setPriority appliquée à une instance de thread change sa priorité d'exécution.
Nous mettons l'instance UnRobinet à la priorité
maximum setPriority(Thread.MAX_PRIORITY) :

Classe Robinet sans changement



Classe Baignoire sans changement



Voici le changement de code dans la classe principale :

```
class RemplirThread {
      public static void main(String[] x){
        Baignoire UneBaignoire = new Baignoire();
        Robinet UnRobinet = new Robinet( );
        UnRobinet.setPriority ( Thread.MAX_PRIORITY );
        UnRobinet.start( );
        UneBaignoire.start( );
      }
}
```

Résultats d'exécution obtenus :

Remplissage, contenu de la baignoire = 50 Remplissage, contenu de la baignoire = 100 Remplissage, contenu de la baignoire = 150 Remplissage, contenu de la baignoire = 200 Remplissage, contenu de la baignoire = 250 Remplissage, contenu de la baignoire = 300 Remplissage, contenu de la baignoire = 350 Remplissage, contenu de la baignoire = 400 Remplissage, contenu de la baignoire = 450 Remplissage, contenu de la baignoire = 500 Remplissage, contenu de la baignoire = 550 Remplissage, contenu de la baignoire = 600	Remplissage, contenu de la baignoire = 600 Fuite, contenu de la baignoire = 580 Remplissage, contenu de la baignoire = 630 Remplissage, contenu de la baignoire = 680 Remplissage, contenu de la baignoire = 730 Remplissage, contenu de la baignoire = 780 Remplissage, contenu de la baignoire = 830 Remplissage, contenu de la baignoire = 880 Remplissage, contenu de la baignoire = 930 Remplissage, contenu de la baignoire = 980 Remplissage, contenu de la baignoire = 1030 **Baignoire enfin pleine !**

---- operation complete.

Nous remarquons bien que le thread de remplissage Robinet a été privilégié dans ses exécutions, puisque dans l'exécution précédente le thread Baignoire-fuite n'a pu exécuter qu'un seul tour de boucle.

Alternons l'exécution de chaque thread :

Nous souhaitons maintenant que le programme alterne le remplissage de la baignoire avec la fuite d'une façon équilibrée : action-fuite/action-remplissage/action-fuite/action-remplissage/...

Nous utiliserons par exemple la méthode **yield ( )** qui cesse temporairement l'exécution d'un thread et donc laisse un autre thread prendre la main. Nous allons invoquer cette méthode yield dans chacune des boucles de chacun des deux threads **Robinet** et **Baignoire,** de telle sorte que lorsque le robinet s'interrompt c'est la baignoire qui fuit, puis quand celle-ci s'interrompt c'est le robinet qui reprend etc... :

Voici le code de la classe Robinet :

```
class Robinet extends Thread {
  public void debite(int quantite){
    for(int i=1; i<100; i++){
      if(Baignoire.contenu.volume < Baignoire.maximum){
        Baignoire.contenu.volume +=quantite;
        System.out.println("Remplissage, contenu de la baignoi
        Thread.yield();
      }
      else {
        System.out.println("Baignoire enfin pleine !");
        break;
      }
    }
  }

  public void run() {
    debite(50);
  }
}
```

Le corps de la méthode main de la classe principale lançant les actions de remplissage de la baignoire reste inchangé :

```
class RemplirThread {
  public static void main(String[] x){
    Baignoire UneBaignoire = new Baignoire();
    Robinet UnRobinet = new Robinet();
    UnRobinet.start();
    UneBaignoire.start();
  }
}
```

Annexe

Vocabulaire pratique : interprétation et compilation en java

Rappelons qu'un ordinateur ne sait exécuter que des programmes écrits en instructions machines compréhensibles par son processeur central. Java comme pascal, C etc... fait partie de la famille des langages évolués (ou langages de haut niveau) qui ne sont pas compréhensibles immédiatement par le processeur de l'ordinateur. Il est donc nécesaire d'effectuer une "**traduction**" d'un programme écrit en langage évolué afin que le processeur puisse l'exécuter.

Les deux voies utilisées pour exécuter un programme évolué sont la **compilation** ou l'**interprétation** :

Un compilateur du langage X pour un processeur P, est un logiciel qui traduit un programme source écrit en X en un programme cible écrit en instructions machines exécutables par le processeur P.

Un interpréteur du langage X pour le processeur P, est un logiciel qui ne produit pas de programme cible mais qui effectue lui-même immédiatement les opérations spécifiées par le programme source.

Un compromis assurant la portabilité d'un langage : une pseudo-machine

Lorsque le processeur P n'est pas une machine qui existe physiquement mais un logiciel simulant (ou interprétant) une machine on appelle cette machine pseudo-machine ou p-machine. Le programme source est alors traduit par le compilateur en instructions de la pseudo-machine et se dénomme pseudo-code. La p-machine standard peut ainsi être implantée dans n'importe quel ordinateur physique à travers un logiciel qui simule son comportement; un tel logiciel est appelé interpréteur de la p-machine.

La première p-machine d'un langage évolué a été construite pour le langage **pascal** assurant ainsi une large diffusion de ce langage et de sa version UCSD dans la mesure où le seul effort d'implémentation pour un ordinateur donné était d'écrire l'interpréteur de p-machine pascal, le reste de l'environnement de développement (éditeurs, compilateurs,...) étant écrit en pascal était fourni et fonctionnait dès que la p-machine était opérationnelle sur la plate-forme cible.

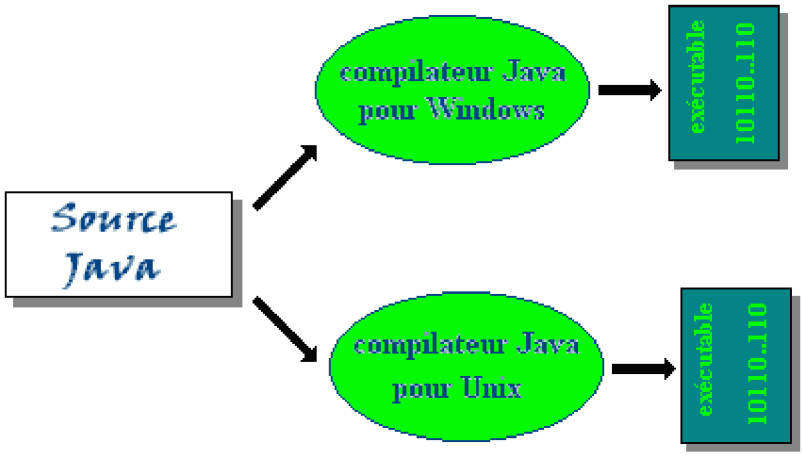
Donc dans le cas d'une p-machine le programme source est compilé, mais le programme cible est exécuté par l'interpréteur de la p-machine.

Beaucoup de langages possèdent pour une plate-forme fixée des interpréteurs ou des compilateurs, moins possèdent une p-machine, **Java est l'un de ces langages**. Nous décrivons ci-dessous le mode opératoire en Java.

Bytecode et Compilation native

Compilation native

La compilation native consiste en la traduction du source java (éventuellement préalablement traduit instantanément en code intermédiaire) en langage binaire exécutable sur la plate-forme concernée. Ce genre de compilation est équivalent à n'importe quelle compilation d'un langage dépendant de la plate-forme, l'**avantage est la rapidité d'exécution des instructions machines par le processeur central.**



Programe source java : xxx.**java** (*portable*)
Programe exécutable sous windows : xxx.**exe** (*non portable*)

Bytecode

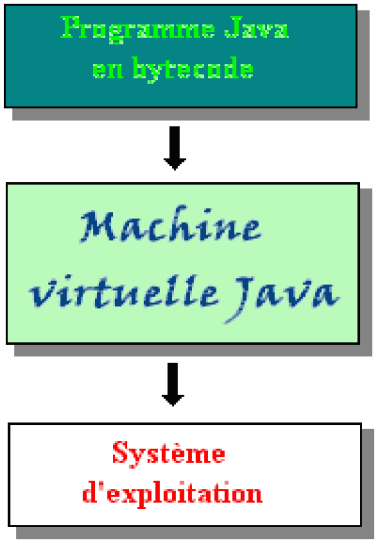
La compilation en bytecode (ou pseudo-code ou p-code ou code intermédiaire) est semblable à l'idée du p-code de N.Wirth pour obtenir un portage multi plate-formes du pascal. Le compilateur **Javac** traduit le programme source xxx.**java** en un code intermédiaire indépendant de toute machine physique et non exécutable directement, le fichier obtenu se dénomme xxx.**class**. Seule une p-machine (dénommée **machine virtuelle java**) est capable d'exécuter ce bytecode. Le bytecode est aussi dénommé **instructions virtuelles java**.

Figure : un programme source ***Exemple.java*** est traduit par le compilateur (dénommé **Javac**) en un programme cible écrit en bytecode nommé ***Exemple.class***

Exemple.java ----------> compilateur Javac ------> Exemple.class

La machine virtuelle Java

Une fois le programme source java traduit en bytecode, la machine virtuelle java se charge de l'exécuter sur la machine physique à travers son système d'exploitation (Windows, Unix, MacOs,...)



Inutile d'acheter une machine virtuelle java, tous les navigateurs internet modernes (en tout cas Internet explorer et Netscape) intègrent dans leur environnement une machine virtuelle java qui est donc installée sur votre machine physique et adaptée à votre système d'exploitation, dès que votre navigateur internet est opérationnel.

Fonctionnement élémentaire de la machine virtuelle Java

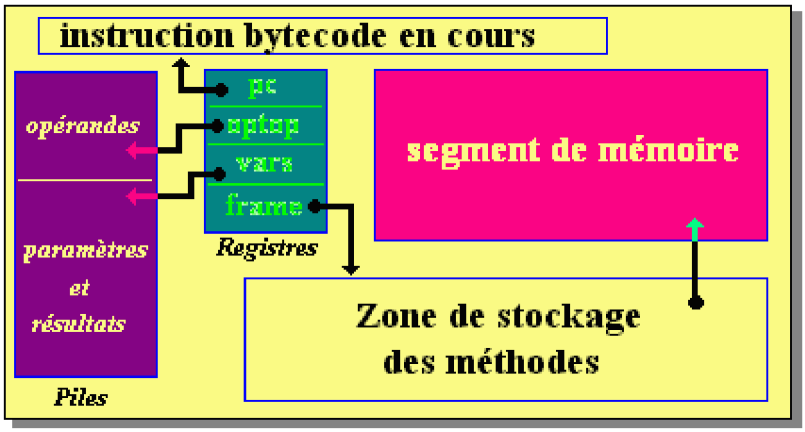
Une machine virtuelle Java contient 6 parties principales

- Un jeu d'instructions en pseudo-code
- Une pile d'exécution LIFO utilisée pour stocker les paramètres des méthodes et les résultats des méthodes
- Une file FIFO d'opérandes pour stocker les paramètres et les résultats des instructions du p-code (calculs)

- Un segment de mémoire dans lequel s'effectue l'allocation et la désallocation d'objets
- Une zone de stockage des méthodes contenant le p-code de chaque méthode et son environnement (tables des symboles,...)
- Un ensemble de registres (comme dans un processeur physique) servant à mémoriser les différents états de la machine et les informations utiles à l'exécution de l'instruction présente dans le registre instruction bytecode en cours.

Comme toute machine la machine virtuelle Java est fondée sur l'architecture de Von Neumann et elle exécute les instructions séquentiellement un à une.

Figure : un synoptique de la machine virtuelle Java



Les registres sont des mémoires 32 bits :

- **vars** : pointe dans la pile vers la première variable locale de la méthode en cours d'exécution.
- **pc** :compteur ordinal indiquant l'adresse de l'instruction de p-code en cours d'exécution.
- **optop** : sommet de pile des opérandes.
- **frame** : pointe sur le code et l'environnement de la méthode qui en cours d'exécution.

JIT , Hotspot

L'interprétation et l'exécution du bytecode ligne par ligne peut sembler prendre beaucoup de temps et faire paraître le langage Java comme "plus lent" par rapport à d'autres langages. Aussi dans un but d'optimisation de la vitesse d'exécution, des techniques palliatives sont employées dans les version récentes des machines virtuelles Java : la technique **Just-in-time** et la technique **Hotspot** sont les principales améliorations en terme de vitesse d'exécution.

***JIT** (Just-in-time)* est une technique de **traduction dynamique durant l'interprétation** que Sun utilise sous le vocable de **compilation en temps réel**. Il s'agit de rajouter à la machine virtuelle Java un compilateur optimiseur qui **recompile localement le bytecode** lors de son chargement et ensuite la machine virtuelle Java n'a plus qu'à faire exécuter des instructions machines de base. Cette technologie est disponible en interne sur les navigateurs de dernière génération.

On peut mentalement considérer qu'avec cette technique vous obtenez un programme java cible compilé en deux passages :

- le premier passage est dû à l'utilisation du compilateur **Javac** produisant du bytecode,
- le second passage étant le compilateur **JIT** lui-même qui optimise et traduit localement le bytecode en instructions du processeur de la plate-forme.

***Hotspot** est une amélioration de **JIT**.*

Un défaut dans la vitesse totale d'exécution d'un programme java sur une machine virtuelle Java équipée d'un compilateur **JIT** se retrouve dans le fait qu'une méthode qui n'est **utilisée qu'une seule fois se voit compilée puis ensuite exécutée**, les mesures de temps par rapport à sa seule interprétation montre que **dans cette éventualité l'interprétation est plus rapide**. La société ***Sun*** a donc mis au point une technologie palliative dénommée **Hotspot** qui a pour but de déterminer dynamiquement quel est le meilleur choix entre l'interprétation ou la compilation d'une méthode. **Hotspot** lancera la compilation des méthodes utilisées plusieurs fois et l'interprétation de celles qui ne le sont qu'une fois.

Bibliographie

Livres papier vendus par éditeur

Livres Java en français

Maurers, Baufeld, Müller & al, Grand livre Java 2, Micro Application, Paris (1999).
Brit schröter, Dossier spécial Java 2 référence, Micro Application, Paris (2000).
A.Tasso, Le livre de Java premier langage, Eyrolles, Paris (2001).
D.Acreman, S.Dupin, G.Moujeard, le programmeur JBuilder 3, Campus press, Paris (1999).
S.Holzner, Total Java, Eyrolles, Paris (2001).
E.&M.Niedermair, Programmation Java 2, Micro Application, Paris (2000).
E.&M.Niedermair, développement Java pour le web, Micro Application, Paris (2000).
M.Morisson & al, secrets d'experts Java, Simon & Scuster MacMillan, Paris (1996).
A.Mirecourt, PY Saumont, Java 2 Edition 2001, Osman-Eyrolles, Paris (2001).
C.Delannoy, Exercices en Java, Eyrolles, Paris (2001)
CS.Horstmann,G.Cornell, au coeur de Java2 notions fondamentales Vol1, Campus press, Paris (2001).
CS.Horstmann,G.Cornell, au coeur de Java2 fonctions avancées Vol2, Campus press, Paris (2002).
H.&P.Deitel, Java comment programmer, Ed. Reynald goulet inc., Canada (2002)
L.Fieux , codes en stock Java 2, Campus press, Paris (2002)
B.Aumaille, J2SE les fondamentaux de la programmation Java, Ed.ENI, Nantes (2002)
R.Chevallier, Java 2 l'intro, Campus press, Paris (2002)
R.Chevallier, Java 2 le tout en poche, Campus press, Paris (2003)
D.Flanagan, Java in a nutshell (trad en fr.), Ed O'REILLY, Paris (2001)
E.Friedmann, Java visuel pro, Ed. First interactive, Paris (2001)
B.Burd Java 2 pour les nuls Ed. First interactive, Paris (2002)
L.Lemay, R.Cadenhead, Java 2 le magnum, Campus press, Paris (2003).
J.Hubbard, structures de données en Java, Ediscience , Paris (2003)
J.Bougeault, Java la maîtrise, Tsoft-Eyrolles, Paris (2003).
D.Barnes & M.Kölling, conception objet en Java avec BlueJ,Pearson education, Paris (2003).

Pour élargir votre horizon Java et développement, un must :

M.Lai, UML et java, InterEditions, 3ème édition, Paris (2004).