Algorithme:

L'algorithmique est la science des algorithmes, visant à étudier les opérations nécessaires à la réalisation d'un calcul.

L'algorithmique doit beaucoup au mathématicien persan Al Kwarizmi (780-850) auteur d'un ouvrage décrivant des méthodes de calculs algébriques. Son nom est d'ailleurs à l'origine du mot algorithme créé par lady Ada Lovelace, fille de lord Byron et assistante de Charles Babbage (1792-1871).

René Descartes en présente une excellente définition dans le Discours de la Méthode :

diviser chacune des difficultés que j'examinerois, en autant de parcelles qu'il se pourroit, et qu'il seroit requis pour les mieux résoudre.

Le nom commun algorithmique est un mot nouveau créé à partir de l'adjectif algorithmique (qui utilise ou se réfère aux algorithmes, par exemple une méthode algorithmique).

Un algorithme est une méthode de résolution de problème énoncée sous la forme d'une série d'opérations à effectuer. La mise en oeuvre de l'algorithme consiste en l'écriture de ces opérations dans un langage de programmation et constitue alors la brique de base d'un programme informatique.

Les informaticiens utilisent fréquemment l'anglicisme *implémentation* (=implantation) pour désigner cette mise en oeuvre.

code source pour désigner le texte, en langage de programmation, constituant le programme.

L'algorithme devra être plus ou moins détaillé selon le niveau d'abstraction du langage utilisé ; autrement dit, une recette de cuisine doit être plus ou moins détaillée en fonction de l'expérience du cuisinier.

Java

```
public class Bonjour
 {
 public static void main(String args[])
 {
 System.out.println("Bonjour tout le monde");
 }
}
```

Remarque Le nom du fichier qui stocke le programme est : “Bonjour.java”, qui a le même nom que la classe.

Compiler et exécuter le Programme:

1. Ouvrir une fenetre MS-DOS
2. Allez vers le répertoire qui contient votre programme “Bonjour.java”
3. Tappez la ligne: javac Bonjour.java
4. Exécuter le programme en tapant:

   java Bonjour

   (il ne faut pas utiliser l’extension .java) et respecter minuscules/majuscules

Les types de données

Type1	Octets	Marge des valeurs
int	4	entier : -2 147 483 648 et 2 147 483 647
short	2	entier : -32 768 et 32 767
long	8	entier: -9 223 372 036 854 775 808 et 9 223 372 036 854 775 807
byte	1	entier: -128 to 127
float	4	réel jusqu'a 10^{38}
double	8	réel jusqu'a 10^{308}
char	2	Un seul caractère. (exp 'a' 'B') 'c' '& ')
boolean	1	Valeur true ou false

```
public class Variable{
public static void main(String args[])
{int a = 34, b = 88, c = 16, sum;
 sum = a + b + c;
 System.out.println("la liste des valeurs :");
 System.out.println(a + ", " + b + " et " + c);
 System.out.println("la somme est: " + sum);
 System.out.println("le produit est: ");
 System.out.println(a * b * c);
 System.out.println("la moyenne est: ");
 System.out.println(sum / 3);
  }
}
```

Les mots réservés de java

abstract	default	goto
new	synchronized	boolean
do	if	package
this	break	double
implements	private	throw
byte	else	import
protected	throws	case
extends	instancesof	public
transient	catch	false
int	return	true
char	final	interface
short	try	class
finally	long	static
void	const	float
native	super	volatile
continue		

Les séquences escape

'\n' (newline)

'\b' (backspace)

'\f' (form feed)

'\'' (single quote)

'\t' (tab)

'\r' (return)

'\\' (backslash)

'\"' (double quote)

Les opérateurs

1. **Postfixe:** . (exp) $exp{+}{+}$ $exp{-}{-}$
2. **Prefixe:** ${+}{+}exp$ ${-}{-}exp$ -exp !exp
3. **Multiplication/division/mod:** * / %
4. **Addition/soustraction:** + -
5. **Shift:** << >> >>>
6. **Comparaison:** < <= > >= == !=
7. **Bit à bit and:** &
8. **Bit à bit xor:** ^
9. **Bit à bit or:** |
10. **Et logique:** &&
11. **Ou logique:** ||
12. **Conditionnel:** *bool_exp* ? *true_val*: *false_val*
13. **Affectation:** =
14. **Operation d'affectation:** += -= *= /= %=

Les entrées/ sorties

Les sorties des informations

```
System.out.print("Bonjour les amis");
System.out.println("Bonjour les amis");
```

Exemple

```
public class TestIo {
 public static void main(String args[]) {
 System.out.print("Bonjour tout le monde");
}
}
```

La saisie des données

x=Lire.i()	int
x=Lire.f()	float
x=Lire.d()	double
x=Lire.b()	byte

Exemple

```
public class TestIo {
 public static void main(String args[]) {
 int x,y;
 System.out.print("Donner x:");
 x=Lire.i();
 System.out.print("Donner y:");
 y=Lire.i();
 System.out.print("La somme de x et y est "+x+y);
}
}
```

Structures de côntroles

1. Le bloc d'intructions (enchainment)
2. Branchement conditionnel
   (a) Si.. alors (**if**)
   (b) Au cas où (**switch**)
3. Répétition (itération)
   (a) tant que ... faire
   (b) faire ... tant que
   (c) pour ... faire

Branchement conditionnel

Si ... alors (**if**)

if (condition)
bloc1

Exemple

```
public class Test {
public static void main(String args[]) {
int x;
System.out.print("Donner x:");
x=Lire.i();
if (x==1)
System.out.print("x est egal a 1");
}
}
```

if (condition)
bloc1
else
bloc2

Exemple

```
public class Test {
public static void main(String args[]) {
int x;
System.out.print("Donner x:");
x=Lire.i();
if (x==1)
System.out.print("x est egal à 1");
else
System.out.print("x est différent de 1");
}
}
```

```
...
int x,y;
System.out.print("Donner x:");
x=Lire.i();
if (x>0)
{ y=x;
  System.out.print("abs. de x est "+y);
}
else
{ y=-x;
  System.out.print("abs. de x est "+y);
}
...
```

$ax + b = 0$

1. $a \neq 0$ alors $x = -b/a$
2. $a = 0$
   (a) $b = 0$ alors éq. indéterminée
   (b) $b \neq 0$ alors pas de solution

```
...
int a,b;
System.out.print("Donner a:");
a=Lire.i();
System.out.print("Donner b:");
b=Lire.i();
if (a!=0)
{ System.out.print("x="+-b/a);
}
else
{ if (b==0)
  System.out.print("éq. indéterminée");
  else
  System.out.print("pas de solution");

}
...
```

Au cas où (**switch**)

switch=un aiguillge limité à *char, byte short, int*

- évalue l'expression qui lui est lieé
- compare au case
- exécute le code du switch si OK
- exécute toutes les instructions des cas suivants
- pour isoler chaque cas, il faut le terminer avec un *break*
- si aucun cas, clause *default*

```
switch (e){
case c1 : b1;
case c2 : b2;
...
case cn : bn;
default: bd
};
```

```
switch (e){
case c1 : b1 break;
case c2 : b2 break;
...
case cn : bn break;
default: bd
};
```

```
...
int i;
System.out.print("Donner i:");
i=Lire.i();
switch(i){
 case 1: System.out.print("I"); break;
 case 2: System.out.print("II"); break;
 case 3: System.out.print("III"); break;
 case 4: System.out.print("IV"); break;
 case 5: System.out.print("V"); break;
 default:  System.out.print("pas traduction");
}
...
```

Répétitions

- tant que ... faire (`while`)
- faire ... tant que (`do ... while`)
- pour ... faire (`for`)

tant que ... faire

while (condition)
    bloc

Fonctionnement

- Tant que l'expression *condition* est vraie l'instruction ou les instructions situées à l'intérieur de la boucle sont exécutées
- Le programme sort de la boucle dès que l'expression *condition* devient fausse
- Une instruction modifiant le résultat du test de sortie de la boucle (*condition*) est placée à l'intérieur de la boucle, de façon à stopper les répétitions au moment souhaité
- Si *condition* est fausse dès le départ, les instructions ne sont jamais exécutées

$$som = \sum_{i=1}^{n} i^2 = 1 + 2^2 + 3^2 + \ldots + n^2$$

```
...
int i,n,som;
System.out.print("Donner n:");
n=Lire.i();
som=0;
i=1;
while (i<=n)
 { som=som+i*i;
   i=i+1;
 }
System.out.print("som="+som);
...
```

```
...
int i,n,som;
System.out.print("Donner n:");
n=Lire.i();
som=0;
i=1;
while (i<=n)
 { som=som+i*i;
   i=i+1;
 }
System.out.print("som="+som);
...
```

i	som
1	0+1=1
2	1+4=5
3	5+9=14
4	14+16=30

faire ... tant que

do

bloc

while (condition);

Fonctionnement

- Les instructions situées à l'intérieur de la boucle sont exécutées tant que l'expression *condition* est vraie
- Les instructions sont exécutées au moins une fois, puisque l'expression conditionnelle est examinée en fin de boucle, après l'exécution des instructions
- si la *condition* reste toujours vraie alors les instruction de la boucle sont répétées à l'infini. On dit que le programme *boucle*
- Une instruction modifiant le résultat du test de sortie de la boucle (*condition*) est placée à l'intérieur de la boucle, de façon à stopper les répétitions au moment souhaité
- un ; est placé à la fin

```
...
int i,n,som;
System.out.print("Donner n:");
n=Lire.i();
som=0;
i=1;
do { som=som+i*i;
     i=i+1;
   }
while (i<=n);
System.out.print("som="+som);
...
```

pour ... faire

for (initialisation ; condition ; incrémentation)
bloc

Fonctionnement

- *initialisation* permet d'initialiser la variable représentant l'indice de la boucle. Ex. `i=1`. Elle est la première instruction exécutée, à l'entrée de la boucle
- *condition* définit la condition à vérifier pour continuer à exécuter la boucle. Ex. `i<=1`. Elle est examinée avant chaque tour de boucle, y compris au premier tour de boucle
- *incrément* est l'instruction qui permet de modifier le résultat de la *condition* en augmentant ou diminuant la valeur de la variable testée

```
...
int i,n,som;
System.out.print("Donner n:");
n=Lire.i();
som=0;
for ( i=1;  i <= n ; i++)
   { som=som+i*i;
   }
System.out.print("som="+som);
...
```

```
...
int i,n,som;
System.out.print("Donner n:");
n=Lire.i();
som=0;
for ( i=1;  i <= n ; som=som+Math.pow(i++,2))
System.out.print("som="+som);
...
```

Rupture

break: quitte la structure dans laquelle elle est comprise

```
...
for(int i=0;i<10;i++)
{ System.out.println(i);
  if (i==4)
    break;
}
System.out.println("salut");

...
```

Exécution du programme

0
1
2
3
4
salut

Continuation

continue:

- interruption du déroulement normal de la boucle
- repris après la derière instruction du corps de la boucle

```
...
for(int i=0;i<5;i++)
{ if (i==3)
    continue;
  System.out.println("i="+i);
}
System.out.println("salut");
 ...
```

Exécution du programme

i=0

i=1

i=2

i=4

salut

Exemple

Écire un programme qui calcule la somme d'une suite de notes. Les notes sont saisis au clavier jusqu'à ce que la note -1 soit entrée.

$$15 \quad 13 \quad 8 \quad 8 \quad 20 \quad 15 \quad -1$$

Programme 1

```
...
int somme=0, note;
System.out.print("Donner une note:");
note=Lire.i();
while (note != -1)
{ somme=somme+note;
  System.out.print("Donner une note:");
  note=Lire.i();
}
System.out.print("la somme est:"+somme);
...
```

$$15 \quad 13 \quad 8 \quad 8 \quad 20 \quad 15 \quad -1$$

Programme 2

```
...
do
{ System.out.print("Donner une note:");
  note=Lire.i();
  if (note != -1)
  somme=somme+note;
} while (note != -1)
System.out.print("la somme est:"+somme);
...
```

Programme 3

```
...
while (true)
{ System.out.print("Donner une note:");
  note=Lire.i();
  if (note == -1) break;
  somme=somme+note;
}
System.out.print("la somme est:"+somme);
...
```

Cascade des conditions

if (condition1)
bloc1
else if (condition2)
bloc2
...
else if (conditionn)
blocn
else
blocn+1

```
...
int i;
System.out.print("Donner i:");
i=Lire.i();
if(i==1)
  System.out.print("I");
else if(i==2)
  System.out.print("II");
else if(i==3)
  System.out.print("III");
else if(i==4)
  System.out.print("IV");
else if(i==5)
  System.out.print("V");
else System.out.print("pas traduction");
}
...
```

Les tableaux

Un **tableau** est une structure qui peut contenir un nombre d'éléments du même type : le **type de base**.

Les éléments sont repérés par un ou plusieurs **indices** de type scalaire.

Le nombre d'éléments du tableau est la **taille** du tableau.

Le nombre d'indices donne la **dimension** du tableau.

Exemple

- Tableau (vecteur) de 10 caractères

$$a \quad b \quad c \quad d \quad e \quad f \quad g \quad h \quad i \quad j \quad k$$

- Tableau à 2 dimensions (matrice) de 3 par 4 entiers

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \\ 11 & 12 & 13 \end{pmatrix}$$

Declaration

type nom[ ];

Exemples:

1. Tableau (vecteur) d'entiers

```
int Tab_Int[];
```

2. Tableau (vecteur) de caractères

```
char Tab_c[];
```

3. Tableau à 2 dimensions (matrice) de réels

```
float m[][];
```

Opérateur d'allocation

Opérateur d'allocation

nom = new type[entier];

Exemples:

1. Tableau (vecteur) de 10 entiers

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

Les éléments du tableau Tab_Int sont:

Tab_Int[0], Tab_Int[1], Tab_Int[2],. . . ,Tab_Int[]

2. Tableau (vecteur) de 20 caractères

```
Tab_c = new char[20];
```

3. Tableau à 2 dimensions (matrice) de 10 par 10 réels

```
m = new float[10][10];
```

Les éléments du tableau m sont:

$$
\begin{array}{cccccc}
m[0][0] & m[0][1] & m[0][2] & \ldots & m[0][2] & m[0][9] \\
m[1][0] & m[1][1] & m[1][2] & \ldots & m[1][2] & m[1][9] \\
\ldots & & & & & \\
m[9][0] & m[9][1] & m[9][2] & \ldots & m[9][2] & m[9][9]
\end{array}
$$

Declaration + allocation

1. Tableau (vecteur) de 10 entiers

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

2. Tableau (vecteur) de 20 caractères

```
char Tab_c[] = new char[20];
```

3. Tableau à 2 dimensions (matrice) de 10 par 10 réels

```
float m[][] = new float[10][10];
```

Tableau de tableaux

•

```
float m[][] = new float[10][10];
```

est équivalent à :

•

```
float m[][] = new float[10][];
for ( int i=0; i<c.length; i++)
   //allocation de chaque ligne de m
   m[i]=new float[10];
```

Saisi des tableaux

```
m = new float[10][12];

for (int i=0; i<10;i++)
 for (int j=0; i<12;i++)
 { System.out.print("donner l'el."+i+" "+j);
   m[i][j]=Lire.()
 }
```

Affichage des tableaux

```
m[][] = new float[10][12];

for (int i=0; i<10;i++)
 { for (int j=0; i<12;i++)
     System.out.print(m[i][j]+"\t");
     System.out.print("\n");
 }
```

Algorithmique

Un algorithme est un procédé de calcul mis en œuvre **pour obtenir un résultat par un nombre fini d'applications d'une règle**.

Ce mot provient du nom latinisé d'un mathématicien arabe du Xe siècle : **Al-Huwarizmi**, originaire de la ville de Huwarism (aujourd'hui Khiva, en Ouzbekistan), et qui écrivit un des premiers traités d'algèbre, où il donnait divers procédés de résolution d'équations.

Un algorithme peut être comparé à une *recette*, avec l'idée supplémentaire de répétition systematique

- jusqu'à ce qu'on ne puisse plus appliquer la recette, ou bien,
- jusqu'à ce que l'on *tourne en rond*.

Cette notion d'itération d'une suite de calculs prend une importance grandissante aujord'hui, où des machines peuvent effectuer sans fatigue, sans erreurs, et en temps très courts, des milliers, voire des millions d'opérations répétitives.

- Recherche dans un tableau
- Recherche dans un tableau trié
- Tri d'un tableau

Recherche dans un tableau

```
int  n,x,tab[];
boolean trouve;
...println("donner la taille du tableau:");
n=Lire.i();
tab = new int[n];

for (int i=0 ; i<n ; i++)
{ ...println("donner l'element "+i);
  tab[i]=Lire.i();
}

...println("donnez l'element recherché:");
x=Lire.i();
```

```
trouve=false;
i=0;
while ( i<n && !trouve )
{ if ( tab[i]==x )
  trouve=true;
  else
  i++;
}

if ( trouve )
...println(x+"trouvé sur la position"+i);
else
...println(x+"n'a pas été trouvé");
```

Recherche dans un tableau trié

- tant que $debut < fin$ et non(TROUVÉ)
  - si l'élémént recherché=tab[m]
    avec $m = \frac{debut+fin}{2}$
    alors TROUVÉ
  - sinon :
    - si l'élémént recherché<tab[m]
      alors rechercher entre $debut$ et $m-1$
    - sinon rechercher entre $m+1$ et fin

```
trouve=false;
debut=0;
fin=n-1;
while ( debut<fin && !trouve)
{ m=Math.round( (debut+fin)/2 );
  if ( tab[m]== x )
  trouve=true;
  else
  { if ( tab[m]>x )
    fin=m-1;
    else
    debut=m+1;
  }
}
if ( trouve )
{ ...print("l'element recherche");
  ...print("a ete trouve en position"+m);
}
else
  ...print("pas trouve");
```

Tri d'un tableau
Tri à bulles

Le tri à bulle permet d'accroitre globalement l'*ordre* à chaque parcour en éliminant, de proche en proche, chacun des *désordres* locaux rencontrés.

- Le tableau est parcouru de son premier à son dernier élément. Chaque fois que l'ordre voulu n'est pas respecté les éléments concernés sont échangés
- Le traitement est alors réitéré sur la tableau amputé de son dernier élément

```
....
dernier=n-1;
encore=true;
while ( encore )
{ encore=false;
  for ( i=0; i<dernier ; i++)
    if (tab[i]>tab[i+1]
    { aux=tab[i];
      tab[i]=tab[i+1];
      tab[i+1]=aux;
      encore=true;
    }
  dernier=dernier-1;
}
```

Fonctions Math

abs	valeur absolue	double, float int, long
acos	Arc cosinus	double
asin	Arc sinus	double
atan	Arc tangente	double
ceil	Arrondi à l'entier supérieur	double
cos	Cosinus	double
exp	Exponentielle	double
floor	Arrondi à l'entier inférieur	double
log	Logarithme naturel (népérien)	double

max	Maximum de deux valeurs	double, float int, long
min	Minimum de deux valeurs	double, float int, long
pow	Puissance a^b	double
random	Nombre aléatoire dans [0,1[	double
rint	Arrondi à l'entier le plus proche	double
round	Arrondi à l'entier le plus proche	long, int
sin	Sinus	double
sqrt	Racine carrée	double
tan	Tangente	double
toDegrees	Conversion de radians en degrés	double
toRadians	Conversion de degrés en radians	double

Manipuler des mots : la classe `String`

- declaration
- saisie
- affichage
- affectation
- méthodes

```
...
String chaine1 = "Bonjour";
String chaine2 = "";

...println("donner chaine2:");
chaine2=Lire.S();

...println(chaine1+chaine2);
```

Recherche de mots et de caractères

- `endsWith()` : Recherche si le mot se termine par le ou les caractères passés en paramètre
- `startsWith()` : Recherche si le mot commence par le ou les caractères passés en paramètre
- `charAt()` : Recherche le caractère placé à la position spécifiée en paramètre : 0...length()-1
- `indexOf()` : Localise un caractère ou une sous-chaîne dans un mot, à partir du début du mot. -1 si la sous-chaîne n'est pas trouvée
- `LastIndexOf()` : ... à partir de la fin du mot
- `substring()` : Extrait une sous-chaîne d'un mot

Comparaison de mots

- `compareTo()` : compare deux mots et retourne :
  - 0 si les deux mots sont identiques (== ne marche pas !)
  - negative si mot1<mot2
  - positive si mot1>mot2
- `equals()` :
  - true si mot1=mot2
  - false si mot1≠mot2
- `equalsIgnoreCase()` : sans différencier les majuscules des minuscules
- `regionMatches()` : portions identiques

Transformation de formats

- `toLowerCase()` : Transforme en minuscules
- `toUpperCase()` : Transforme en majuscule
- `concat()` : concatène
- `replace()` : remplace le caractère donné en premier argument par le caractère donné en deuxième argument
- `length()` : calcule la longueur de la chaîne

Fonctions

- Appeler une fonction
- Définir une fonction

Appeler une fonction :

```
....
double resultat, a;
...println("donner a:");
a=Lire.d();
resultat=Math.sqrt(a);
 ...println("le résultat est "+resultat);
```

Périmètre d'un cercle

```
....
double rayon, resultat;
...println("donner le rayon:");
rayon=Lire.d();
resultat=perimetre(rayon);
 ...println("le résultat est "+resultat);
```

Definir une fonction

- déterminer les instructions composant la fonction
- associer le nom de la fonction aux instructions
- établir les parametrès de la fonction
- préciser le type du résultat fourni par la fonction

L'en-tête d'une fonction :

- le nom de la fonction
- les paramètres
- le type du résultat

```
public static double perimetre (double r)
{ double p;
  p=2*Math.PI*r;
  return p;
}
```

```
public class Cercle
{  public static void main (String [] arg)
   {
     ...
   }

public static double perimetre (double r)
  {
    ...
  }
}
```

```
public class Cercle
{  public static void main (String [] arg)
   { double rayon, resultat;
     ...println("donner le rayon:");
     rayon=Lire.d();
     resultat=perimetre(rayon);
    ...println("le résultat est "+resultat);
   }

   public static double perimetre (double r)
   { double p;
     p=2*Math.PI*r;
     return p;
   }
}
```

Fonctions

Intérêts des fonctions

- *meilleure lisibilité et concision* des programmes (on évite d'écrire plusieurs fois la même suite d'instructions)
- *niveau d'abstraction plus élevé* lors de l'analyse (un nom remplacera par exemple une action complexe qui sera détaillée plus tard), ce qui facilite **l'analyse descendante** (du plus globale au plus spécifique)
- *modifications plus aisées* : elles ne concernent qu'une partie du programme
- *partage possible* d'une fonction par plusieurs programmes; on peut donc constituer des **bibliothèques** de fonctions
- *gain de place mémoire*, car le code de la fonction n'apparaît qu'une seule fois en mémoire

Définition et appel des fonctions

- Définir une fonction
- Appeler une fonction

Définition

La définition d'une fonction se fait en indiquant

- *son identificateur* qui permettra aux programmes appelants de la désigner
- *la liste des paramètres* (éventuels) qui doivent lui être fournis et le type de chacun d'eux. Lors de la définition on parle de **paramètres formels**
- *le type (éventuel) de la valeur renvoyée* par la fonction (`void` si la fonction ne renvoie pas de valeur)
- *la description* de l'action réalisée par la fonction

L'appel

C'est le moment où une fonction est effectivement utilisée par une autre fonction. L'instruction d'appel d'une fonction se résume tout simplement à l'identification de la fonction suivie de la liste (éventuelle) des **paramètres effectifs**. Elle déclanche l'exécution des instructions de la fonction.

Une correspondence stricte sera donc nécessaire entre la liste des paramètres formels et paramètres effectifs sur :

- le nombre des paramètres
- l'ordre d'apparition des paramètres dans la liste
- le type de chaque paramètre

Exemple

maxi() qui fournisse en résultat la plus grande des deux valeurs données en paramètres :

```
public class Maximum
{  public static void main (String [] arg)
   { int x,y,sup;
     ...println("donner le x:");
     x=Lire.i();
     ...println("donner le y:");
     y=Lire.i();
     sup=maxi(x,y);
    ...println("le max est "+sup);
   }

   public static int maxi(int a, int b)
   { int m=a;
     if(b > m) m=b;
     return m;
   }
}
```

Récursivité

La notion de définition récursive en informatique correspond à celle de définition par récurrence en mathématiques.

Exemple:

1. $n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$

$$n! = \begin{cases} 1 & \text{if} \quad n = 0, \\ (n-1)! \times n & \text{if} \quad n > 0 \end{cases}$$

2. Relation *est parent de* x est parent de y si :

   - Soit y est père, mère, fils ou fille de x
   - Soit il existe z tel que x est parent de z et z est parent de y

3. Recherche d'un objet x dans un tableau t

   - Recherche x dans la première moitié de t
   - Recherche x dans la deuxième moitié de t

Fonction récursive

Une fonction est **récursive** si sa définition contient un (ou pusieurs) appel(s) à elle-même.

Intéret et inconvénients de la récursivité

- Concision des algorithmes
- Analyse plus naturelle de certaines problèmes qui se prêtent plus au raisonnement récursif, et dont la résolution par une méthode itérative serait beaucoup plus complexe
- Moins de variable locales nécessaires
- L'exécution d'un programme récursif demande plus de place mémoire et plus de temps a cause de la répétition des appels

Mise en œuvre de la récursivité

Pour vérifier la finitude (être sur que le programme se termine au bout d'un nombre fini d'appels) un programme récursif devra présenter deux parties :

- Un cas de base dans laquelle il n'est pas fait d'appel récursif
- Un cas général qui contient un ou plusieurs appels récursifs mais sur des arguments de *taille* de plus en plus petite jusqu'à atteindre le cas de base.

D'une manière générale, un algorithme récursif aura la forme suivante :

```
si <condition>
Alors <bloc 1> (cas de base)
Sinon <bloc 2> (cas general)
```

```
public class Factorielle
{  public static void main (String [] arg)
   { int n,f;
     ...println("donner le n:");
     n=Lire.i();
     f=fact(n);
    ...println(n+"!="+f);
   }

   public static int fact (int a)
   { if(a==0) return 1;
     else return a*fact (a-1);
   }
}
```

La structure d'un programme Java

1. Un programme contient :
   - des déclarations de variables
   - une fonction principale, appelée `main()`
   - un ensemble de fonctions définies par le programmeur
2. Les fonctions contiennent
   - des déclarations de variables
   - des instructions élémentaires (affectation, test, répétitions, ...)
   - appels à des autres fonctions (prédéfinies ou non)
3. Chaque fonction est comparable à une boîte noire, dont le contenu n'est visible en dehors de la fonction

```
public class NomDeLaClasse
{  // déclarations de variables
   public static void main (String [] arg)
   { // déclarations de variables
     // instructions élémentaires
     // appels à des autres fonctions
   }

   public static type NomDeLaFonction(paramètres)
   { // déclarations de variables
     // instructions élémentaires
     // appels à des autres fonctions
   }
}
```

La visibilité des variables

```
public class Visibilite
{  public static void main (String [] arg)
   { int valeur=2;
     ...println("valeur= "+valeur);
     modifier();
     ...println("valeur= "+valeur);
   }

   public static void modifier()
   {  valeur=3;
   }
}
```

La variable `valeur` est invisible dans la fonction `modifier()`

Variable locale à une fonction

```
public class VariableLocale
{  public static void main (String [] arg)
   { int valeur=2;
     ...println("valeur= "+valeur);
     modifier();
     ...println("valeur= "+valeur);
   }

   public static void modifier()
   {  int valeur=3;
      ...println("valeur= "+valeur);
   }
}
```

Il y a une variable `valeur` à l'intéreur de la fonction `main()` et une autre variable `valeur` à l'intéreur de la fonction `modifier()`

Variable de classe

```
public class VariableDeClasse
{  static int valeur;
   public static void main (String [] arg)
   { valeur=2;
     ...println("valeur= "+valeur);
     modifier();
     ...println("valeur= "+valeur);
   }

   public static void modifier()
   {  valeur=3;
      ...println("valeur= "+valeur);
   }
}
```

La variable `valeur` est définie pour tout le programme `VariableDeClasse`

Le véritable nom de la variable de classe

```
public class VeritableNom
{  static int valeur;
   public static void main (String [] arg)
   { VeritableNom.valeur=2;//valeur=2;
     ...println("valeur= "+valeur);
     modifier();
     ...println(VeritableNom.valeur);
   }

   public static void modifier()
   { int valeur=5;
     VeritableNom.valeur=3;
   }
}
```

Le nom d'une variable de classe est constitué du nom de la classe, suivi d'un point puis du nom de la variable déclarée.

Comment les fonctions communiquent

- par des variables de classe
- par passage de paramètres par valeur
- par le resultat d'une fonction

$$f(x) = 3 \times x$$

Communication par des variables de classe

```
public class Variables
{  static int valeur;
   public static void main (String [] arg)
   { ...println("donner la valeur:");
     valeur=Lire.i();
     tripler();
     ...println(valeur);
   }

   public static void tripler()
   { valeur=3*valeur;
   }
}
```

Communication par passage de paramètres par valeur

```
public class Parametres
{  public static void main (String [] arg)
   { int valeur;
     ...println("donner la valeur:");
     valeur=Lire.i();
     tripler(valeur);
     ...println(valeur);
   }

   public static void tripler(int valeur)
   { valeur=3*valeur;
     ...println(valeur);
   }
}
```

Communication par le resultat d'une fonction

```
public class Resultat
{  public static void main (String [] arg)
   { int valeur;
     ...println("donner la valeur:");
     valeur=Lire.i();
     valeur= tripler(valeur);
     ...println(valeur);
   }

   public static int tripler(int valeur)
   { return 3*valeur;
   }
}
```

Définir des classes

classe=

- données
- méthodes (fonctions)

Définition de données à l'aide d'instructions de déclaration de variables et/ou d'objets. Ces variables sont de type simple (`int, char, float`) ou de type composé (`String` ...)

Ces données décrivent les informations caractéristique de l'objet que l'on souhaite définir. Elle sont appelées *champ*, *attribut* ou *membre* de la classe.

Construction des méthodes Ce sont les méthodes associées aux données, et sont définies par le programmeur. Elles se construisent comme de simples fonctions. Ces méthodes représentent tous les traitements et comportements de l'objets que l'on cherche à décrire.

```
public class Cercle
{ public int x, y; // position du centre
  public int r; // rayon
  public void afficher() // affichage
  {   .println(" centre en "+x+","+y);
      .println(" de rayon "+r);
  }
  public double perimetre ()
  { return 2*Math.PI*r;
  }
  public void deplacer (int nx, int ny)
  { x=nx;
    y=ny;
  }
  public void agradir (int nr)
  { r=r+nr;
  }
}
```

```
public class FairDesCercles
{  public static void main(String [] arg)
   {  Cercle A = new Cercle ();
      A.afficher();
       .println(" entrez la position en x");
      A.x=Lire.i();
       .println(" entrez la position en y");
      A.y=Lire.i();
            .println(" entrez le rayon");
      A.y=Lire.i();
      A.afficher();
      double p=A.perimetre();
       .println(" le perimetre est "+p);
      A.deplacer(5,2);
       .println(" apres le deplacement :");
      A.afficher();
      A.agrandir(10);
       .println(" apres agrandissement :");
      A.afficher();
   }
}
```