ALGORITHMIQUE ET PROGRAMMATION ORIENTEE OBJET

I. INTRODUCTION

1.1. HISTORIQUE

1991: Concepts Java pour Sun
- -conception d'un langage pour des appareils électroniques(code embarqué)
- -syntaxe proche du C++
- -concepts de machine virtuelle du langage Pascal.

1995: Logiciel HOTJAVA:un navigateur web écrit en Java par Sun.

1996: Version 1.01 et 1.02

.

.

2002: Version 1.4:disponible sur http://java.sun.com

Plusieurs kits de développement logiciel Java2:J2SDK

1.2. PROGRAMMATION ORIENTEE OBJET (POO)

La POO est différente de la programmation structurée (ex: Pascal): les programmes sont découpés en fonctions indépendantes qui sont elles-mêmes constituées de blocs d'instructions structurées.

PROGRAMME = STRUCTURE DE DONNEES + ALGORITHMES

Les structures de données et les fonctions qui les utilisent sont séparées.
Les données sont généralement en début de programme et donc visibles de toutes les fonctions qui suivent.

- PROBLEMES LIES A CETTE ORGANISATION:

-effets de bords (état inattendu des données dans une fonction suite à leurs modifications par une autre fonction).
-nécessité de reconstruire le programme lorsque les structures de données sont modifiées.

- SOLUTIONS: POO

-repose sur plusieurs principes : -l'objet
-l' encapsulation des données
-la classe
-l'héritage
-le polymorphisme

1.2.1. L'objet:

Un programme POO construit des objets.
Chaque objet associe des données (aussi appelées champs, variables, attributs) et des méthodes (rarement appelées procédures, fonctions) agissant exclusivement sur les données de l'objet.

1.2.2. L'encapsulation des données:

L'encapsulation permet de protéger les données d'un objet.
En effet, il n'est pas possible d'agir directement sur les données d'un objet. Il est nécessaire de passer par ses méthodes qui jouent le rôle d'interface obligatoire.
L'appel de méthode est en fait l'envoi d'un message à l'objet.
Vu de l'extérieur, un objet se caractérise uniquement par les spécifications de ses méthodes .
La manière dont sont réellement implantées les données est sans importance.
Le fonctionnement interne de l'objet est caché au monde extérieur.

1.2.3. La classe :

La classe est l'élément fondamental contenant les éléments de programme: une déclaration de données, une méthode ou une instruction appartenant à une classe.
Une classe est déclarée avec le mot-clé *class*.
Le concept de classe correspond à la généralisation de la notion de type des programmes structurés.

Définition :

Une classe est une description d'un ensemble d'objets ayant une structure de données commune et disposant des mêmes méthodes.
Les définitions des classes doivent correspondre à des unités fonctionnelles indépendantes.
Une classe est un modèle d'objet. A l'inverse, un objet a une réalité matérielle car il possède des champs avec valeurs.

Remarque:

Une classe statique ne permet pas de construire d'objets.
Les objets apparaissent alors comme des variables d'un tel type classe: un objet est une instance de sa classe (un objet possède un type classe).
Les valeurs des champs sont propres à chaque objet.
Les méthodes sont communes à l'ensemble des objets d'une même classe.

JAVA possède des bibliothèques de classe appelées API JAVA (Application Programmation Interfaces, Interface de Programmation d'Application).

PROGRAMMER EN JAVA = PROGRAMMER SES PROPRES CLASSES + UTILISATION DES API

- Avantages de créer ses propres classes:
  - -connaissance exacte de son fonctionnement.

- Avantages d'utiliser des classes API ou des classes disponibles:
  - -gain de temps (pas de conception, pas de développement)
  - -performance
  - -portabilité

1.2.4. L'héritage :

L'héritage permet de définir une nouvelle classe (une sous-classe) à partir d'une classe existante (super-classe, classe-mère) à laquelle on ajoute de nouvelles données, de nouvelles méthodes.
L'héritage permet donc de spécialiser des classes anciennes parfaitement au point:
-économie du code.
-structuration des fonctionnalités d'un programme (lisibilité).

L'héritage peut-être réitéré:
La classe C hérite de la classe B qui elle-même hérite de la classe A.
L'héritage s'applique aussi bien à ses propres classes qu'aux classes API.
Les classes sont organisées en hiérarchie.

JAVA NE PERMET PAS L' HERITAGE MULTIPLE (héritage de plusieurs super-classes) mais les méthodes d'interface permettent de traiter de façon simple des situations similaires.

1.2.5. Le Polymorphisme:

Il permet à des méthodes (des fonctions) différentes dans des classes différentes d'avoir le même nom.
Il permet d'économiser des identificateurs de fonctions (lisibilité).

1.2.6. Le langage de POO presque pur:

JAVA est un langage de POO: un programme est formé d'au moins une classe dans laquelle des objets sont instanciés.

JAVA est un langage presque pur:
-existence de types primitifs pour les booléens, les entiers, les réels, les caractères.

Les valeurs correspondantes ne sont pas des objets même si en général ces types primitifs sont utilisés pour définir les champs d'une classe, donc finalement un objet.
-existence de méthodes particulières appelées *méthodes statiques de classe* (déclarées avec le mot-clé *static*) qui sont utilisables de façon indépendante d'un objet.

Comme ces méthodes peuvent déclarer localement des variables d'un type primitif, on retrouve les possibilités de procédure et fonction d'un langage de programmation structuré.
Une telle méthode appelée *main* joue le rôle de programme principal.
-encapsulation des données non absolue.

1.3: JAVA et la programmation évènementielle:

- Interface console: une seule fenêtre avec un petit nombre de fonctionnalités (déplacement, fermeture, agrandissement,...)

- Interface graphique: fenêtre principale s'ouvrant au lancement du programme.

JAVA permet des applications (graphisme sans web) et des applets (graphisme sur le web) en utilisant pratiquement les mêmes fonctionnalités graphiques.

1.4: JAVA et la portabilité:

- Classiquement:

Un programme est portable car un même code source peut être exécuté sur différentes machines après une nouvelle compilation.

- En JAVA:

La compilation d'un code source produit, non pas des instructions machines, mais un code intermédiaire formé de byte codes.
D'une part, ce code est exactement le même quelque soit le compilateur et la machine concernés. D'autre part, ces byte codes sont exécutables sur toute machine disposant du logiciel d'interprétation appelé machine virtuelle JVM (JAVA Virtual Machine).

1.5: D'autres particularités : (pas fait)

1.6: Phases de développement d'un programme JAVA:

- **PHASE 1: EDITION:**

-éditeur:- vi et emacs sous UNIX
-Bloc-Notes sous WINDOWS
-Environnement de dvpt Intégré (EDI)

Le nom de fichier d'un programme JAVA se termine toujours par l'extension *.java*

Exemple: *Programme.java*

- **PHASE 2: COMPILATION:**

-La commande du compilateur JAVA pour compiler un programme JAVA et le traduire en byte codes est *javac*.
-La compilation génère un fichier possédant le même nom que la classe et contenant les byte codes avec l'extension *.class*.
Le compilateur génère un fichier compilé pour chaque classe. Ainsi, si le fichier source contient plusieurs classes, alors plusieurs fichiers ont l'extension *.class*.

Exemple: *javac Programme.java* génère un fichier *Programme.class*

Mettre l'extension à la suite du nom en respectant la casse du nom du fichier.
JAVA est sensible à la casse.

PHASE 3: CHARGEMENT:

Une application peut être chargée est exécutée par la commande de l'interpréteur JAVA: *java*

Exemple: *java Programme* (pas d'extension *.class* à la suite du nom)

Une applet peut être chargée est exécutée par :
-un chargeur de classe lancé par le navigateur Web

-ou par la commande appletviewer du J2SDK

Exemple : *appletviewer Programme.html*

PHASE 4: VERIFICATION

PHASE 5: EXECUTION

II.GENERALITES

2.1.:Programmation écriture console:

Problème: Ecriture d'un texte dans une fenêtre console.
Fichier: *Ecriture.java*

```
public class Ecriture
        {
        public static void main(String[]args)
                {
                System.out.println("Un programme Java");
                }
        }
```

Execution: Un programme JAVA

Une classe publique, déclarée avec le mot-clé *public*, est une classe qui est accessible à n'importe quelle autre classe.

Un fichier source peut contenir plusieurs classes à condition qu'une seule classe au plus, soit publique. Cette classe publique est la classe contenant le *main* .

Le code source d'une classe publique doit toujours se trouver dans un fichier portant le même nom et possédant l'extension *.java*.
Le nom du fichier source et le nom de la classe doivent avoir la même casse.

Caractère d'échappement : \

Séquence de contrôle = caractère d'échappement + caractère spécial

Exemple : “\n “ :nouvelle ligne
“\r “ :retour chariot
“\f “ :nouvelle page
“\t “ :tabulation
“\ \“ : \
“\ ‘ “ : ‘
“\ “ “ : “

2.2. : PROGRAMME ECRITURE FENETRE :

Problème : Ecriture d’un texte dans une fenêtre graphique
Fichier : *EcritureFenetre.java*

```
Import javax.swing.* ;
Public class EcritureFenetre
{
Public static void main ( String[] args)
        {
        JoptionsPane.showMessageDialog(null,”Fenetre Java”);
        }
}
```

Exécution: Message

JAVApossède de nombreuses classes prédéfinies. Ces classes sont regroupées en catégories de classes de même genre appelées *package (modules).* Une référence générale aux packages existe sous l’appellation de bibliothèques de classes Java ou d’interface de programmation d’application Java(API ou d’applications programming interface).
Tous les packages de l’API Java sont stockés dans un répertoire *java ou javax* qui contiennent de nombreux sous-répertoires.
Un nom de package est associé à un répertoire.
La classe *JoptionPane* dans le package *javax.swing* permet d’afficher une boîte de dialogue avec des informations.

Ligne : *import javax.swing.**

L’instruction *import* permet d’identifier et de charger les classes nécessaires au programme.

La classe *JoptionPane* appartient au *package javax.swing* qui comporte de nombreuses classes permettant de définir *des interfaces utilisateur graphique ou GUI* (Graphical User Interface)

2 solutions :

-importer uniquement la classe nécessaire du package *javax.swing : javax.swing.** et chargement par le compilateur des classes effectivement utilisées au programme.

-invoquer toutes les classes du package *javax.swing : javax.swing.** et chargement par le compilateur des classes effectivement utilisées par le programme.

*Javax.swing.** ne concerne que les classes du répertoire *javax.swing*
Les classes des sous-répertoires de *javax.swing* ne sont pas invoquées. Pour les invoquer, il faut préciser le sous-répertoire associé, par exemple *javax.swing.event.**

Problème : Ecriture d'un texte dans une *applet*
Fichier : *EcritureApplet.java*

```
import java.awt.* ;
import javax.swing.* ;

public class EcritureApplet extends Japplet
        {
        public void paint (Graphics g)
                {
                g.drawString("APPLET JAVA", 100,100) ;
                }
}
```

Execution:

2.3. : Programmation LECTURE:

Problème : Lecture d'un texte d'une fenêtre console
Fichier : *Lecture.java*

```
Import java.io.* ;
//Méthode de lecture au clavier
        public class Lecture
        {
//Lecture d'une chaîne
        public static String lireString()
        {
        String ligne_lue=null;
        try
                {
                InputStreamReader lecteur=new InputStreamReader(System.in);
```

```
        BufferedReader entree=new BufferedReader (lecteur);
        Ligne_lue=entree.readLine();
        }
catch (IOException err)
        {
        System.exit(0);
        }
return ligne_lue;
}
}
```

//Lecture d'un réel double

```
public static void double lireDouble()
{
double x=0 ;
try
        {
        String ligne_lue=lireString() ;
        x=Double.parseDouble(ligne_lue) ;
        }
catch (NumberFormatException err)
        {
        System.out.println("Erreur de données");
        System.exit (0) ;
        }
return x ;
}
```

Ce type de lecture peut ou non mal se passer: pour cela, on utilise une construction de type *"try... catch"*

//Lecture d'un entier

```
public static int LireInt()
{
int n=0 ;
try
        {
        String ligne_lue=lireString() ;
        n=Integer.parseInt(ligne_lue) ;
        }
catch (NumberFormatException err)
        {
        System.out.println("Erreur de données");
        System.exit (0) ;
        }
return n;
}
```

// Programme test de la classe Lecture

```
public static void main(String[] args)
{
System.out.print("Donner un double:");
Double x ;
x=Lecture.lireDouble()
System.out.println(« Résultat »+x) ;            /*+x sert à faire une conversion de
                                                  type*/
System.out.print(« Donner un entier ») ;
int n ;
n=Lecture.lireInt() ;
System.out.println(« Résultat »+n) ;
```

<u>Exécution :</u> Donner un Double : 10.01
Résultat 10.01
Donner un entier :10
Résultat 10

Le symbole d'affectation est '='

L'opérateur + possède une propriété intéressante :
Dès que l'un des opérateurs est de type chaîne, l'autre opérateur est converti en chaîne.
-pour les variables de type primitif, la conversion en chaîne de caractères est automatique.
-pour les objets, la conversion en chaînes fait appel à la méthode *toString()* de l'objet.
L'opérateur + dans *System.out.println* permet la concaténation de chaînes conduisant à l'impression d'une chaîne de caractères.
La *méthode print()* diffère de *println()* par l'absence de saut de ligne en fin d'impression.

2.4. : Règles générales d'écriture :

<u>2.4.1 : Identificateurs :</u>

Dans un langage de programmation, *un identificateur est une suite de caractères* pour désigner les entités dans un programme :variables, méthodes, objets, classes,...
C'est une suite de lettres et de chiffres (sans espace), le premier caractère étant une lettre.

<u>Par convention :</u>

-les identificateurs de classe commencent **toujours par une majuscule.**
- les identificateurs de variables et de méthodes commencent toujours par une minuscule.
-les identificateurs formés par la concaténation de plusieurs mots comportent une majuscule à chaque début de mot sauf pour le premier mot qui dépend du type de l'identificateur.

Exemple : *public class ClasseNouvelle*

2.4.2 : Mots-Clés

boolean, catch, continue, else, false, int, package, …

2.4.3: Séparateurs

C.f. manuel JAVA

2.4.4.: Format libre:

JAVApermet *le format libre* :

Une instruction s'étend sur plusieurs lignes et une même ligne peut comporter plusieurs instructions

Le point-virgule marque la fin de l'instruction.

Par convention : une instruction par ligne

Une instruction peut être :

-un bloc d'instruction

-l'instruction vide :le point-virgule uniquement

-une instruction de contrôle

-une expression suivie du point-virgule, une expression étant soit un opérateur d'affectation(=), soit un opérateur de création d'objet(new), soit un opérateur d'incrémentation ou de décrémentation, soit un appel à une méthode .

2.4.5. : Commentaires

3 formes :

Commentaire usuel : pouvant s'étendre sur plusieurs lignes : /*…*/
Commentaire de fin de ligne : s'arrêtant en fin de ligne : //
Commentaire de documentation: /**…*/ : pouvant être extraits automatiquement par des programmes utilitaires de création de documentation tels que Javadoc qui génère automatiquement une documentation au format HTML.

2.4.6 : Blocs :

Bloc délimité par les accolades *{ et }*. Il comporte :

-aucune, une ou plusieurs déclarations de variables locales (connues que dans le bloc)

- aucune, une ou plusieurs instructions

2.4.7. : Code Unicode

Java utilise le système Unicode pour coder des caractères. Chaque caractère Unicode est codé sur *2 octets conduisant à 2^16=65536 possibilités* qui permettent de représenter la plupart des alphabets et des symboles mathématiques et techniques.

Le code source est toujours actuellement édité avec un éditeur générant des caractères codés sur 1 octet.

Il existe peu d'éditeurs de texte Unicode. Mais le compilateur Java traduit le fichier source en Unicode avant de générer les byte codes.
Ainsi, même les caractères Unicode du code source ne sont pas portables, il sont représentés de façon portable dans les byte codes.

La notation Java permettant d'introduire un caractère Unicode dans le code source est réalisé avec les séquences d'échappement Unicode qui n'utilisent que des caractères ASCII *: \uxxxx* où *u est le symbole pour Unicode*, *x est un chiffre hexadécimal et xxxx* *désigne le caractère ayant comme Unicode la valeur xxxx.*

Il faut 4 chiffres hexadécimaux pour représenter des caractères codés sur *16 bits (16^4)= 2^16.*

<u>Remarque :</u> Certains éditeurs de programme source permettent l'utilisation des caractères nationaux qui sont les caractères Unicode.

III. TYPES PRIMITIFS

3.1. : Notion de type :

Les types primitifs permettent de manipuler des booléens, des entiers, des réels, des caractères.

Ils sont les seuls types du langage qui ne soient pas des classes. Ils sont utilisés pour définir les champs de données de toutes les classes.
La déclaration de variables est obligatoire avant leur utilisation.
Dans une déclaration de variables, le type des variables est suivi de la liste des identificateurs de variables séparés par des virgules.

En JAVA, les déclarations peuvent apparaître à n'importe quel endroit du programme. De préférence, la déclaration de la variable se fait au moment de sont utilisation.
<u>Remarque:</u> Les champs des objets possèdent une initialisation implicite.

3.2. :Type booléen :

<u>Déclaration d'une variable booléenne :</u> *boolean test*
Une variable booléenne prend deux valeurs : *false* et *true*.
<u>Affectation d'une variable booléenne</u> : *test =(n<m)*

3.3. : Type entier :

Le type entier permet de représenter de façon exacte une partie des nombres entiers relatifs :

TYPE	TAILLE (en Octets)	VALEUR MINIMALE	VALEUR MAXIMALE
Int	4	Integer.MIN_VALUE	Integer.MAX_VALUE
long	8	Long.MIN_VALUE	Long.MAX_VALUE

<u>*Par défaut:*</u> une constante entière est de type int
Une constante entière suffixée par *I ou L est de type long*

<u>Exemple :</u> 452L

Une constante peut-être exprimée en base 8(octale) en faisant précéder la valeur par 0

<u>Exemple :</u> 014(8+4=12 en base 10)

Une constante entière peut-être exprimée en base 16 (hexadécimale) en faisant précéder la valeur par 0x ou 0X

<u>Exemple :</u> 0X1A

3.4. :Type réel :

Le type réel permet de représenter de façon approchée une partie des nombres réels.

Ainsi, pour tester l'égalité de deux nombres réels, il est préférable de comparer la valeur absolue de leur différence à un nombre très petit.

TYPE	TAILLE (en Octets)	PRECISION	VALEUR MINIMALE	VALEUR MAXIMALE
Float	4	7	Float.MIN_VALUE	Float.MAX_VALUE
double	8	15	Double.MIN_VALUE	Double.MAX_VALUE

<u>*Par défaut:*</u> une constante réelle est *de type double*.
Une constante réelle peut s'écrire en notation décimale ou exponentielle.

<u>Exemple :</u> Notation décimale : 1.1 ; 1. ; .1
Notation exponentielle : $1.1^{E}10$; 1.E10 ; $.1^{E}10$

Une constante réelle suffixée par *f ou F est de type float*.
L'affectation d'une constante réelle de type double à une variable de type float nécessite *une conversion de type ou transcryptage (cast).*
La conversion d'un objet, d'une variable ou d'une constante dans un type donné se fait en les précédant du type placé entre parenthèses.

Exemple :
float x ;
x=(float)12.5 ; // et non x=12.5 ;

Problème : Ecriture d'une variable réelle en utilisant un formatage
Fichier : EcritureRéel.java

```
import java.awt.* ;
public class EcritureReel
{
public class static void main (String[] args)
        {
        double x=10.123456789;
        System.out.println("x= "+x);

//au moins un chiffre à gauche du point décimal
//2chiffres exactement à droite du point décimal

        DecimalFormat deuxDecimal = new DecimalFormat («0.00 ») ;
        System.out.println(« x= « +deuxDecimal.format(x)) ;
        }
}
```

Execution : x=10.123456789
x=10.12

3.5. : Type caractère :

Déclaration d'une variable caractère : *char c*.
Une constante caractère est notée entre apostrophe.
Exemple : 'a'

3.6. : Initialisation :

Toute variable doit être initialisée avant d'être utilisée (sinon erreur de compilation)
Une variable peut être initialisée lors de sa déclaration par une constante ou une expression autre qu'une constante, par exemple par une instruction de lecture.
Exemple : int n=10 ; n=Lecture.lireInt()

Les situations de variables partiellement initialisées sont détectées à la compilation, par exemple dans une structure conditionnelle *si*.
Mais l'initialisation d'une variable dans une structure conditionnelle *si-sinon* , n'entraîne pas d'erreur de compilation.
Exemple :
int n ;
if (…) n=0 ;
…=n ; // erreur de compilation

Exemple :
int n ;
if (…) n=0 ;
else n=0 ;
…=n ; // pas d'erreur à la compilation

3.7. :Constante :

Une constante est une variable initialisée dont la valeur ne peut être modifiée dans la suite du programme en utilisant le *mot-clé final* (valeur connue à la compilation) *Une telle constante est appelée constante symbolique*.

Exemple :
final int N=10

Le mot-clé final peut être utilisé avec une expression d'initialisation de la valeur (valeur non connue à la compilation mais à l'exécution)

Exemple :
final int N=2*m

Les mots-clés static et public sont souvent associés au *mot-clé final*.
Une constante déclarée de type *static* appartient à la classe (sa place dans la classe et non dans l'objet).

Exemple :
static final int N=10

Une constante *déclarée publique* est connue en dehors de la classe.

Exemple : public final int N=10
Les constantes *Pi et e* sont accessibles par Math.Pi et Math.E (champs statiques)

Par convention :

Les identificateurs de constantes sont obligatoirement en majuscule. Il est toujours préférable d'utiliser des constantes symboliques plutôt que des constantes littérales.
En effet, pour changer la valeur d'une constante symbolique, il suffit de changer sa valeur dans sa déclaration .
Par ailleurs, il est difficile de localiser toutes les utilisations des constantes sans erreur, car une même valeur peut avoir des significations différentes.

3.8. : Expression constante :

Le mot-clé final peut être utilisé avec une expression constante (valeur connue à la compilation)

Exemple :
final int M=10
final int N=2*m

IV. Opérateurs et expressions.

4.1. : Originalité des notions d'opérateurs et d'expressions.

Une expression est une suite d'opérandes et d'opérateurs.
Dans les langages classiques :
-l'expression possède une valeur mais ne réalise aucune action, en particulier, elle j'affecte pas de valeur à une variable.
-l'affectation donne une valeur à une variable : réalise donc une action.

En JAVA, les notions d'expression et d'affectation ne sont pas disjointes :
-des opérateurs d'incrémentation particuliers par exemple *i++*, intervenant au sein d'une expression, conduisent à la réalisation d'une action.
-une affectation par exemple, i=10 peut être considérée comme une expression puisque le symbole est un opérateur.

En JAVA, l'instruction est une expression se terminant par un point- virgule (instruction expression)

4.2. : Opérateurs arithmétiques :

4.2.1. : Opérateurs unaires :

+ et -

4.2.2 :Opérateurs binaires :

+ , - , * , / , %(modulo)

Les opérateurs ne sont définis qu 'avec des opérandes de même type.
L'opérateur % peut porter sur des entiers positifs et négatifs mais également sur des réels positifs et négatifs.
Il n'existe pas d'opérateur puissance :
Il faut utiliser la *méthode statique Math.pow.*
La class Math est une classe dérivée de *java.lang* qui est la classe chargée par défaut.

Pow(double a, double b) pour a^b

4.2.3. : Priorité des opérateurs :

Les priorités des opérateurs en JAVA sont celles de l'algèbre.

4.3. : Conversions implicites dans les expressions :

Une expression mixte est une expression avec des opérandes de type différent.

Hiérarchie des conversions d'ajustement de type :

Int ->long ->float ->double

Les conversions sont effectuées en considérant un à un les opérandes concernés.

Remarque : Il n’existe pas de conversion implicite de double en float.

Promotions numériques :

La promotion numérique est la conversion systématique d’un type apparaissant dans une expression en int sans considérer les types des autres opérandes.

{byte, short,char} ->int

Exemple :
char c;
c+1 ;
|
int : valeur numérique de c

Comportement en cas d’exception pour le entiers : cf. manuel

Comportement en cas d’exception pour le réels :
-aucune opération sur les réels, même pas une division par 0, ne conduit à un arrêt d’exécution. Mais, il existe des valeurs représentant l’infini :
-infinity avec les constantes Double.POSITIVE_INFINITY et Float.POSITIVE_INFINITY
- infinity avec les constantes Double.NEGATIVE_INFINITY et Float.NEGATIVE_INFINITY
-NaN (Not A Number) avec les constantes Double.NaN et Float.NaN

4.4. : Opérateurs relationnels (comparaison)

Ils sont : < ; <= ; > ;>= ; ==(égal) et !=(différent)
Les valeurs représentant infini peuvent être comparées avec les valeurs réelles.

Exemple :
double x=10^{E}10
if (x=Double.POSITIVE_INFINITY …) : résultat est vrai

La comparaison des caractères est basée sur le code Unicode avec les relations suivantes

‘0’<’1’<…<’9’<’A’<’B’<…<’Z’<’a’<’b’<…<’z’

4.5. : Opérateurs logiques :

4.5.1 : Opérateur unaire :

! (négation)

4.5.2. : Opérateurs binaires :

&(et), |(ou), ^(ou exclusif), &&(et avec court-circuit) et ||(ou avec court-circuit)

Avec les deux derniers opérateurs, le deuxième opérande n'est évalué que si sa valeur est nécessaire pour décider si l'expression est vraie ou fausse.
Cette propriété est indispensable dans certaines constructions comme les tableaux (cas d'une condition comportant simultanément des tests sur l'indice et les éléments d'un tableau)

Avec l'opérateur &&, la condition dépendante doit être placée après l'autre condition.
Dans les expressions utilisant && et si les conditions sont indépendantes des unes des autres, il faut placer à gauche la condition susceptible d'être fausse.
Dans les expressions utilisant || et si les conditions sont indépendantes des unes des autres, il faut placer à gauche la condition susceptible d'être vraie.

4.6. : Opérateur d'affectation :

Il impose que son premier opérande soit une référence à un emplacement dont la valeur peut être modifiée. Par exemple, une variable non déclarée avec le mot-clé final.

Exemple : i=10

Cet opérateur possède une associativité de droite à gauche.

Exemple : j=i=10

Il a une priorité plus faible que les opérateurs arithmétiques et relationnels.
Les conversions implicites dans l'affectation sont :

Byte -> short -> int -> long -> float -> double
char -> int -> long -> float -> double

4.7.: Opérateurs d'incrémentation et de décrémentation :

Ces opérateurs unaires portant sur une variable, conduisent à des expressions qui possèdent une valeur et qui réalisent une action.

L'opérateur d'incrémentation ++ est :
-un opérateur de pré-incrémentation quand il est placé à gauche de son opérande
-un opérateur de post-incrémentation quand il est placé à droite de son opérande

L'opérateur de décrémentation - - est :

- un opérateur de pré-décrémentation quand il est placé à gauche de son opérande
- un opérateur de post-décrémentation quand il est placé à droite de son opérande

Exemple :
y=x++ y=x ; x=x+1
y=++x x=x+1 ; y=x

Lorsque seul importe l'effet d'incrémentation (resp. décrémentation), l'opérateur ++ (resp. --) peut être indifféremment placé avant ou après la variable.
Ainsi : i++ ++i
Il n'est pas possible d'utiliser les opérateurs d'incrémentation et de décrémentation avec des expressions autre qu'une valeur

Exemple :
(n+1)++ : FAUX

4.8. : Opérateurs d'affectation élargie :

Ils permettent de condenser les affectations de la forme :

Variable= variable opérateur expression
en Variable opérateur = expression

Liste des principaux opérateurs d'affectation élargie : += -= *= /= %=

Exemple : a+=b a=a+b
Les opérateurs relationnels et logiques ne sont pas concernés par cette possibilité.

4.9. :Opérateur cast :

L'opérateur *cast* permet de forcer la conversion d'une expression quelconque dans un type donné.

Exemple : Soient n et p , 2 variables de type int
(double)(n/p)

Il existe autant d'objets de *cast* que de types différents, y compris les types classes. La conversion d'un objet d'une classe dérivée en un objet de classe de base est implicite. La conversion inverse, ie d'un objet d'une classe de base en un objet de classe dérivée est possible avec l'opérateur de *cast*.

La priorité d'un opérateur de *cast* est élevée. Il est nécessaire de placer entre parenthèses l'expression concernée.
La conversion par l'opérateur de *cast* ne produit jamais d'erreur d'exécution.

4.10 : Opérateurs de manipulations de bits :

cf. manuel

4.11. : Opérateur conditionnel :

L'opérateur conditionnel est un opérateur ternaire (3opérandes) permettant de traduire :

si (expression_test) alors variable=expression_1
sinon variable= expression_2

par

variable =(expresssion_test) ?expression_1 :expression_2

Exemple : x=(a>b) ?a :b

Cette instruction affecte à x, la plus grande des valeurs de a et b.
Il est également possible que l'expression conditionnelle soit évaluée sans que la valeur soit utilisée.

Exemple : (a>b) ?i++:i- -

V. Instructions de contrôle.

5.1. :Instruction if :

L'instruction *if* présente deux formes :

If (expression_booléenne) instruction_1
[else instruction_2]

Exemple :
If (++i<max)… i=i+1 ; if (i<max)…
If (I++<max)… i=i+1 ; if (I-1<max)… =>conséquence: 1 boucle en plus.

Un *else* se rapporte toujours au dernier *if* rencontré auquel un *else* n'a pas encore été attribué.

5.2 : Instruction switch :

Est une instruction de sélection multiple

```
switch (expression)
        {
 case cste_1 : [suite_instructions_1]
 …………………………..
case cste_n : [suite_instructions_n]
[default : suite_instructions]
        }
```

expression : est une expression de type byte, short, int ou char ; les expressions de type byre, short ou char ne peuvent être que de simples variables (sauf avec l'opérateur cast) ; le type long n'est pas autorisé.

Cste_i : expression constante d'un type compatible par affectation avec le type de expression.

Suite_instructions_i : suite d'instructions quelconques.
Si expression vaut cste_i, suite_instructions_i est exécutée et toutes les suites d'instructions suivantes peuvent être hiérarchisées.
Pour sortir du *switch* immédiatement, les suites d'instructions doivent se terminer par un *break.*

5.3. : Instruction while :

de la forme

while (expression_booléenne) instruction

répète l'instruction tant que la condition de poursuite est vraie.
La condition est testée avant chaque parcours de la boucle.
Cette condition doit être initialisée.

```
……………………..
int compteur=1 ;
while (compteur<=10)                    compteur : 1,….10
{
instruction_unique ;
compteur++ ;
}
……………………
```

…………………..
int compteur=0 ;
while (++compteur<=10) instruction_unique ; compteur :1,….,10
…………………

---

int compteur=0 ;

while (compteur++<=10) compteur : 1,…..,11
……………………………………

Choisir en priorité des indices de boucles de type entier.

5.4. : Instruction do…while :

do (instruction) while (expression_booléenne);

L'instruction *do… while* répète l'instruction tant que la condition de poursuite est vraie. Une telle boucle est parcourue au moins une fois.

5.5. : Instruction for :

for ([initialisation] ;[expression_booléenne] ;[incrémentation])

instruction
initialisation : est une déclaration ou une suite d'expression quelconques séparées par des virgules. La déclaration est locale au bloc.
incrémentations : sont une suite d'expression quelconques séparées par des virgules.

Problème : Exemple d'une instruction *for*
Fichier : InstructionFor.java

```
Public class InstructionFor
{
public static void main (String[ ] args)
                {
                for (int i=1,j=1;(i<=5); I++,j++)
                        {
                        System.out.println ("I="+I+" j="+j);
                        }
                }
```

}

Execution : i=1 j=1
i=2 j=3
i=3 j=6
i=4 j=10
i=5 j=15

Soit le programme partiel suivant:

```
………………………..
for (int i=1 ; i<=100 ; i+=2) j+=i ;
……………………….
```

```
……………………..
for (int i=1 ; i<=100 ; j+= i, i+=2) ;
……………………….
```

5.6.: Instruction break:

ne doit être utilisée que dans une instruction *switch*

VI. Classes et objets .

6.1. : CLASSES :

La notion de classe généralise la notion de type. La classe comporte des champs (données) et des méthodes.
La notion d'objet généralise la notion de variable. Un type classe donné permet de créer (d'instancier) un ou plusieurs objets du type, chaque objet comportant son propre jeu de données.

6.1.1. : Définition d'une classe :

Une classe contient un en-tête et un corps comportant les définitions de ses champs et méthodes.

```
Class Equation
{
// Définition des champs et méthodes de la classe.
```

…………………
}

En l'absence du mot-clé *public*, l'accès à la classe *Equation* est limité aux seules classes du même package.

(i) Définition des champs :

```
private double coeffX2 ;
………………………..
```

Le mot-clé *private* précise que les champs de la classe *Equation* ne sont pas accessibles à l'extérieur de la classe, ie en dehors de ses propres méthodes.

encapsulation des données.

(ii) Définition des méthodes :

-initialisation
-résolution
-affichage

Définition méthode= similaire à la méthode main

```
public void initialisation (double X2, double X1, double X0)
{
coeffX2=X2 ;
…………………
}
```

6.1.2. : Utilisation d'une classe :

La classe Equation permet d'instancier des objets de type Equation et de leur appliquer les méthodes publiques initialisation, résolution et affichage.
Elle ne peut être utilisée directement. Il faut créer une variable de type *Equation* (uneEquation) appelée *instance de la classe Equation*.

Cette utilisation se fait dans une autre méthode quelconque, en particulier la méthode main.

La déclaration est similaire à une variable de type primitif…

Equation uneEquation

…réserve un emplacement mémoire pour une référence à un objet de type *Equation* (pas de réservation pour un emplacement mémoire pour un objet de type Equation)
Dit autrement, l'identificateur d'objet *uneEquation* est une référence à l'objet et non une variable contenant directement une valeur, comme pour les variables de type primitif.

La réservation d'un emplacement mémoire pour un objet de type *Equation* ou allocation, se fait en appelant l'opérateur unaire *new Equation()* qui fournit en résultat la référence à cet emplacement.

Ainsi, l'affectation

UneEquation=new Equation()

Créée (construit) un objet de type *Equation* et place sa référence dans *uneEquation* qui doit avoir été déclarée au préalable.
L'opérateur *new* fait appel à *Equation()* qui est le constructeur par défaut de la classe *Equation* .

Il est possible de regrouper déclaration et création.

Equation uneEquation = new Equation()

Après avoir déclaré et créé un objet, il est possible d'appliquer les méthodes à l'objet correspondant, ie référencé par *uneEquation* .

Une Equation.initialisation (1.0,5.0,1.0)

Cette instruction appelle la méthode initialisation du type *Equation* en l'appliquant à l'objet de référence *uneEquation* et en lui transmettant les arguments 1.0, 5.0, 1.0 ,…

Le préfixe *uneEquation* précise à la méthode sur quel objet elle doit opérer. Ainsi, l'instruction coeffX2=X2 de la méthode initialisation() place la valeur reçue pour X2 dans le champ coeffX2 de l'objet *uneEquation* .

L'appel d'une méthode se fait en préfixant le nom de la méthode par le nom de l'objet suivi d'un point :

NomObjet.nomMéthode (liste_arguments)

Une méthode est toujours suivie de *()* (permettant la distinction avec un champ)
Avec une méthode utilisant des arguments , la liste des arguments réels séparés par des virgules est mise entre *()*
Par convention, l'objet *uneEquation* est appelé une variable de classe.

6.1.3. : Plusieurs classes dans un même fichier source :

Un fichier source peut contenir plusieurs classes mais obligatoirement une seule classe publique.
La classe contenant la méthode *main* doit obligatoirement être publique (pour l'accès à la machine virtuelle).

Une classe peut avoir 2 attributs d'accès :
-publique
-sans attribut (accès de package) : classe reste accessible à toutes les classes du même package, donc en particulier du même fichier source.

La compilation créée un fichier *.class* par classe.

Problème : Résolution d'une équation du 2nd degré
Fichier : EquationSecondDegre.java

```
Import java.text.DecimalFormat ;

        //Classe Equation
class Equation
{
        //Les coefficients et les racines sont sous forme de champs.
private double coeffX2 ;
private double coeffX1 ;
private double coeffX0 ;
private double racine1 ;
private double racine2 ;







        //Méthode d'initialisation
public void initialisation(double X2, double X1, double X0)
{
coeffX2=X2 ;
.
.
.
}



        //Méthode résolution
public void résolution()
{
double discri ;
discri=(coeffX1*coeffX1-4*coeffX2*coeffX0) ;
racine1=……… ;
racine2=……… ;
}


        //Méthode d'affichage
public void affichage()
{
DecimalFormat deuxDecimal=new DecimalFormat(« 0.00 ») ;
System.out.println(« Racine1= »+deuxDecimal.format(racine1) ;
```

```
System.out.println(« Racine2= »+deuxDecimal.format(racine2) ;
}
}


        //Classe Test
public class EquationSecondDegre
{
        //Méthode principale
public static void main (String[ ] args)
{
Equation uneEquation;                    // Déclaration de l'objet uneEquation
uneEquation=new Equation() ;
uneEquation.initialisation(1.0,5.0,1.0);
uneEquation.résolution();
uneEquation.affichage();
}
}
```

Exécution : Racine1= -0,21
Racine2= -4,79

6.1.4. : Une classe par fichier source :

(1)Sauvegarder la classe Equation dans un fichier Equation.java

Fichier : Equation.java

```
import java.test.DecimalFormat ;

public class Equation
{
…….
}
```

(2)Compiler le fichier Equation.java. Génération d'un fichier Equation.class
(3) Sauvegarder la classe EquationSecondDegre dans un fichier EquationSecondDegre.java
(4)Compiler le fichier EquationSecondDegre.java

6.2. :CONSTRUCTEURS

6.2.1. : Principe :

La méthode d’initialisation permet d’attribuer des valeurs aux champs d’un objet de type *Equation* . Cette méthode doit donc être impérativement appelée.

Le constructeur permet :
-d’automatiser le mécanisme d’initialisation
-d’attribuer des valeurs initiales mais également des actions

Le constructeur est une méthode portant le même nom que la classe et sans valeur de retour

Transformation de la méthode d’initialisation de la classe *Equation* ....

```
    //Méthode d'initialisation
public void initialisation(doubleX2, double X1, doubleX0)
{
coeffX2=X2 ;
coeffX1=X1 ;
coeffX0=X0 ;
}
```

........en un constructeur :

```
    //constructeur
public Equation (double X2, double X1, double X0)
{
coeffX2=X2 ;
coeffX1=X1 ;
coeffX0=X0 ;
}
```

Transformation de l’appel de la méthode d’initialisation de la classe Equation..........

```
    //Appel de la méthode d'initialisation
Equation uneEquation ;

UneEquation=new Equation() ;

UneEquation.initialisation(1.0,5.0,1.0);
```

..........en un appel du constructeur

```
	//appel du constructeur
Equation uneEquation ;

UneEquation=new Equation(1.0,5.0,1.0) ;
```

Remarque:
L'instruction uneEquation=new Equation() est **impossible** (erreur de compilation) car la classe Equation dispose d'un constructeur qui, dans cet exemple a besoin de 3 arguments réels.
Il existe une solution avec le constructeur vide.

6.2.2. :Quelques règles :

Un constructeur porte obligatoirement le nom de la classe qu'il instancie.
Par définition, un constructeur ne fournit aucune valeur. Aucun type ne doit figurer dans son en-tête, même pas le type void.
Si par erreur un type est donné à un constructeur, on créé une méthode et non un constructeur.

Une classe peut n'avoir aucun constructeur (constructeur par défaut sans argument également appelé constructeur vide implicite).
Par exemple :

Equation une Equation = new Equation()

Une classe peut avoir plusieurs constructeurs.
Quand une classe possède au moins un constructeur, un constructeur par défaut ne peut plus être utilisé.
Le constructeur par défaut ne se distingue pas d'un constructeur sans argument.
Un constructeur ne peut pas être appelé directement depuis une autre méthode.

EquationConstructeur une Equation ;
UneEquation.EquationConstructeur(1.0,5.0,1.0) ; //impossible

Un constructeur peut appeler un autre constructeur de la même classe en utilisant le *mot-clé « super ».*
Un constructeur peut être déclaré *privé* : il ne peut plus être appelé de l'extérieur de la classe et donc, ne peut être utilisé pour instancier des objets.
Cette possibilité n'a d'intérêt que si la classe possède au moins un autre constructeur public faisant appel à ce constructeur privé.

6.2.3. : Création d'un objet :

La création d'un objet entraîne toujours par ordre chronologique les opérations suivantes :

-initialisation par défaut de tous les champs par objet.
-initialisation explicite des champs de l'objet.
-exécution des instruction du corps du constructeur.

```
	//Constructeur
public Equation (double X2, double X1, double X0)
{
system.out.println (« coeffX2= »+coeffX2) ;
coeffX2=X2 ;
system.out.println(« coeffx2= »+coeffX2) ;
```

Exécution : coeffX2=0.0
coeffX2=1.0

Initialisation par défaut des champs d'un objet :

Type boolean :false
Type char : caractère de code nul
Type entier :0
Type réel (double, flottant) : 0. et 0.f
Type Objet (champ d'une classe référençant un objet) : null

Initialisation explicite des champs d'un objet :

Comme pour une variable locale, une initialisation de champ peut comparer une constante ou une expression courante.

Exécution des instructions du corps du constructeur :

Les initialisations des champs doivent être effectuées dans le constructeur (fonction du constructeur).
La fonction d'un constructeur est de réaliser l'allocation dynamique de la mémoire nécessaire à la création de l'objet qui est réalisé à l'exécution.
Par opposition, l'allocation statique de la mémoire est réalisée à la compilation.

Cas des champs déclarés avec l'attribut *final* :

Comme une variable locale, un champ peut être déclaré avec l'attribut *final* afin d'imposer une seule initialisation.
Alors qu'une variable peut être initialisée tardivement, n'importe où dans une méthode, un champ déclaré avec l'attribut final doit être initialisé au plus tard par le constructeur. Par ailleurs, il n'existe pas d'initialisation par défaut, il faut donner une valeur.

6.2.4. : Initialisation avec les champs d'un objet :

Comme pour les méthodes, pour utiliser les champs d'un objet, il suffit de préfixer le nom du champ par le nom de l'objet suivi par le champ :

NomObjet.nomChamp

Fortement déconseillé !! (ne respecte pas l'encapsulation des données)

6.3 . : CONCEPTION DES CLASSES :

Les méthodes de classe sont de 3 types :
-les constructeurs
-les méthodes d'accès(accessor) de la forme getxxx qui fournissent des informations relatives à l'état d'un objet, ie des valeurs de certains champs sans les modifier :

```
        //Classe Equation
class Equation
{
 public double getRacine1()
 {
 return racine1;
 }



public double getRacine2()
 {
 return racine2;
 }
…
}

        //Classe test
public class EquationSecondDegre
{
//Méthode principale
public static void main (String[ ] args)
{
……
System.out.println ("sol1= "+uneEquation.getRacine1();
System.out.println ("sol2= "+uneEquation.getRacine2();
}
}
```

Exécution
Sol1= -0.20871215252208009
Sol2= -4.1912878474779195

-Les méthodes d'altération (mutation) de la forme setxxx, qui modifient l'état d'un objet, ie les valeurs de certains champs :

```
        //Classe Equation
class Equation
{
public void setRacine1(double rac1)
{
racine1=rac1;
}

public void getRacine1(double rac1)
{
racine2=rac2;
}
…….
}
```

```
        //Classe Test
public class EquationSecondDegre
{
        //Méthode principale
public static void main (String[ ] args)
{
…
uneEquation.affichage();
uneEquation.setRacine1(1.0);
uneEquation.setRacine2(2.0);
uneEquation.affichage();
}
}
```

Exécution : Racine1= -0,21 Racine1=1.0
Racine2=-4,79 Racine2=2.0

6.4. :AFFECTATION ET COMPARAISON D'OBJETS :

Soient deux variables E1 et E2 de type Equation.

Equation E1, E2 ;

Soient 2 objets E1 et E2 de type Equation.

E1=new Equation (1.0,5.0,1.0) ;
E2=new Equation (2.0,6.0,2.0);

Soient 3 variables E1, E2, E3 de type Equation.

Equation E1, E2, E3 ;

Soient 2 objets E1 et E2 E1 et E2 de type Equation

E1=new Equation (1.0,5.0,1.0) ;
E2=new Equation (2.0,6.0,2.0);

Les instructions E3=E1 , E1=E2 , E2=E3 conduisent aux affectations suivantes:
-la variable E1 désigne l'objet Equation (2.0,6.0,2.0)
-les variables E2 et E3 désignent le même objet Equation(1.0,5.0,1.0)

Il est impossible de définir une variable locale d'un type primitif sans l'initialiser. Cette règle d'initialisation s'applique également aux variables locales de type classe.

```
        //Classe Test
public class EquationSecondDegre
{
        //Méthode principale
public static void main (String[ ] args)
{
Equation uneEquation;
        //uneEquation=new Equation (1.0,5.0,1.0);
uneEquation.résolution();
………..
}
}
```

Un champ d'un objet est toujours initialisé :
-soit implicitement à une valeur nulle, sauf les champs avec attribut *final* qui doivent recevoir explicitement une valeur
-soit explicitement (lors de la déclaration)
-soit au niveau du constructeur.

Cette règle s'applique également aux champs de type classe (objets) pour lesquels cette valeur nulle correspond à une valeur particulière de référence notée par le mot-clé *null.*

Une telle référence *null* ne désigne aucun objet.

Alors qu'une référence non définie est détectée à la compilation (cf exemple ci-dessous), une référence null n'est détectée qu'à l'exécution.

```
        //Classe Test
public class EquationSecondDegre
{
        //Méthode principale
public static void main (String[ ] args)
{
Equation uneEquation=null;
        //uneEquation=new Equation (1.0,5.0,1.0);
uneEquation.résolution();
………..
}
}
```

Cette référence *null* peut être utilisée dans une comparaison.

```
        //Classe Test
public class EquationSecondDegre
{
        //Méthode principale
public static void main (String[ ] args)
{
Equation uneEquation=null;

        if (uneEquation==null)
        System.out.println("Référence null");
………..
}
}
```

Tant qu'un objet n'a pas été créé par new, la référence sur cet objet vaut *null*. La référence *null* ne pointe sur aucun objet. Une tentative d'accès à un objet non créé provoque la levée d'une exception de type *NullPointerException*.
Il est parfois nécessaire d'initialiser des références à *null.*

L'affectation de variables de type objet se limite à la recopie de références. Elle n'entraîne pas la recopie de la valeur des objets.

Les opérateurs *== et !=* portent sur les références des objets et non pas sur les valeurs de leurs champs. Leur intérêt est limité.
<u>Exemple:</u> Pour comparer 2 objets de type chaîne, il faut comparer le contenu, donc utiliser la méthode Equals().

6.5. : RAMASSE-MIETTES :

L’opérateur de création d’un objet est *new*.
Il n’existe pas d’opérateur de destruction d’un objet, comme on en dispose en Pascal.
Java utilise un mécanisme de gestion automatique de la mémoire.
Possibilité de créer un objet sans référence.

Avec la méthode principale :

```
        //Méthode principale
public static void main (String[ ] args)
{
Equation uneEquation=new Equation (1.0,5.0,1.0);

uneEquation.résolution();
………..
}
}
```

La référence uneEquation est supprimée.

```
        //Méthode principale
public static void main (String[ ] args)
{
(new Equation (1.0,5.0,1.0)).résolution();
………..
}
}
```

Cet objet non référencé devient candidat au ramasse-miettes.

6.6. : PROPRIETES DE METHODES :

6.6.1. : Méthodes ne fournissant aucun résultat :

Méthode avec le *mot-clé void* dans son en-tête.
Méthode appelée : Objet.méthode (liste arguments) ;

6.6.2. : Méthodes fonction fournissant un résultat :

Méthode avec le type du résultat dans son en-tête.

```
        //Méthode somme
```

```
public double somme()
{
double som ;                         //double som=0.0 ;
som=racine1+racine2 ;

return som ;
}
```

Méthode appelée :

```
        //Méthode principale
public static void main (String[ ] args)
{
…………
System.out.println(« Somme= «+uneEquation.somme()
}
……
```

Possibilité d'utiliser le résultat d'une méthode fonction dans une expression de la classe Equation.

```
        //Classe Equation
classe Equation
{
……
private double sommeSquare=0.0 ;
………..
        //Méthode somme
public double somme()
{
…
}
```

<u>6.6.3. : Arguments formels et effectifs :</u>

<u>Arguments formels (muets) :</u> arguments dans l'en-tête de la définition d'une méthode.

La valeur est fournie à la méthode lors de son appel (contrairement à une variable locale à la méthode).
Possibilité de déclarer son argument formel avec un attribut *final*. Dans ce cas, la valeur de cet argument ne peut plus être modifiée par la méthode.

<u>Exemple :</u> cf. //Méthode d'initialisation//

Utiliser des arguments effectifs de même type que les arguments formels.

Eviter les possibilités de conversion implicite légale.

Java respecte l'ordre des arguments.
Eviter la dépendance des arguments f(n++,n) et f(n, n++) conduisent à des appels différents.

Arguments effectifs : arguments fournis lors de l'appel de la méthode. Ces arguments peuvent être des expressions (la transmission se fait par valeur).

6.6.4. : Variables locales :

La portée d'une variable locale est limitée au bloc de la méthode où elle est déclarée.
Une variable locale ne peut pas avoir le même nom qu'un argument formel.
L'emplacement d'une variable locale est alloué à l'entrée de la méthode et est libéré à la sortie de la méthode (variable non rémanente).

Une variable locale est obligatoirement d'un type primitif ou d'un type classe. Une variable locale de type classe contient la référence à un objet.
Toute variable locale, y compris une référence à un objet, doit être impérativement initialisée avant d'être utilisée.
Il est possible de définir des variables locales à un bloc : for, if, bloc artificiel, etc., …

6.7. : CHAMPS ET METHODES DE CLASSE :

6.7.1. : Champs de classe (Champs Statiques) :

Un champ de classe est un champ unique pour toutes les instances d'une même classe (champ global partagé par toutes les instances de classe). Le champ de classe existe indépendamment de tout objet de sa classe.
Un champ de classe est déclaré avec l'attribut *statique.*
L'appel d'un champ de classe se fait en précisant le nom du champ par le nom de la classe suivi d'un point :

NomClasse.nomChamp

Problème : Champ statique pour compter le nombre de résolution :

```
Class Equation
{
…..
private static int nbRésolution=0 ;

//Constructeur.
Public Equation(double X2, double X1, double X0)
{
…
nbRésolution++ ;
}
```

```
//Méthode de résolution
public void résolution()
{
…
}

//Méthode affichage
public void affichage()
{
system.out.println(« nbRésolution= »+nbRésolution() ;
…
}
}
```

```
//classe Test
public class EquationSecondDegre
{
//Méthode principale
public static void main (String[ ] args)
{
Equation uneEquation = new Equation(1.0,5.0,1.0);
UneEquation.résolution();
UneEquation.affichage;
Equation uneEquation = new Equation (2.0,6.0,2.0);
….
```

<u>Exécution:</u>

nbRésolution=1
Racine1=-0,21
Racine2=-4,79
NbRésolution=2
Racine1=-0,38
Racine2=-2,62

Même programme sans l'attribut static :

```
Class Equation
{
…
private int nbRésolution=0 ;
……
}

//Classe test
public class EquationSecondDegre
{
…..
}
```

EXECUTION:

```
nbRésolution=1
Racine1=-0,21
Racine2=-4,79
NbRésolution=2
Racine1=-0,38
Racine2=-2,62
```

6.7.2. : Méthodes de classe (méthodes statiques) :

Une méthode de classe est une méthode indépendante de tout objet d'une classe. Une méthode de classe agit uniquement sur les champs de classe et non sur les champs non statiques.
Une méthode de classe est déclarée avec l'attribut statique.

L'appel d'une méthode de classe se fait en préfixant le nom de la méthode par le nom de la classe suivi d'un point :

NomClasse.nomMéthode.

Problème : Méthode statique pour compter le nombre de résolution :

```
Class Equation
{
…..
private static int nbRésolution=0 ;

//Constructeur.
Public Equation(double X2, double X1, double X0)
{
…
nbRésolution++ ;
}

//Méthode de résolution
public void résolution()
{
…
}

//Méthode de nbSolution
public static int nbSolution()
{
return nbRésolution ;
}

//Méthode affichage
public void affichage()
{
system.out.println(« nbRésolution= »+nbRésolution() ;
```

```
…
}
}
//classe Test
public class EquationSecondDegre
{
//Méthode principale
public static void main (String[ ] args)
{
Equation uneEquation = new Equation(1.0,5.0,1.0);
UneEquation.résolution();
UneEquation.affichage;
Equation uneEquation = new Equation (2.0,6.0,2.0);
System.out.println(« nbSolution= »+Eq.nbSolution()) ;
….
```

Exécution:

nbRésolution=1
Racine1=-0,21
Racine2=-4,79
NbRésolution=2
Racine1=-0,38
Racine2=-2,62

Les méthodes objets peuvent utiliser des champs statiques.
De préférence, les méthodes qui n'opèrent que sur des champs statiques doivent être déclarées statiques.

Utilisation :
Les méthodes de classe sont utilisées pour :
-le comptage des objets d'une classe.
-la description de la classe
-le regroupement de fonctionnalités, par exemple la classe Math.

6.7.3. : Initialisation des champs de classe :

L'initialisation des champs usuels (non statiques) est faite à la création d'un objet de la classe en 3 étapes :
-initialisation par défaut (int i ;)
-initialisation explicite éventuelle lors de la déclaration (int i=0 ;)
-initialisation par le constructeur.

L'initialisation des champs statiques est faite avant la création d'un objet de la classe puisqu'un champ statique ne dépend pas des objets de la classe. Cette

initialisation se fait donc soit à la déclaration d'un objet de la classe , soit lors de l'appel d'une méthode statique de la classe et se limite donc aux 2 étapes :

-initialisation par défaut

-initialisation explicite éventuelle lors de la déclaration

Un champ statique avec l'attribut *final* doit obligatoirement recevoir une valeur initiale au moment de sa déclaration puisqu'il ne peut être initialisé implicitement, ni être initialisé par un constructeur.

Les champs statiques et les méthodes statiques sont alloués à l'exécution.

Bloc d'initialisation statique :

Dans la définition d'une classe, il est possible d'introduire un ou plusieurs blocs d'instructions précédés du *mot-clé static*.

```
static
{
……..
}
```

Ce bloc statique permet de faire des initialisations de champs statiques qui ne peuvent pas être faites en une seule expression.
En effet, il n'est pas possible d'utiliser un constructeur qui n'est pas concerné par l'initialisation des champs.

Les instructions d'un bloc statique n'ont accès qu'aux champs statiques de la classe.

6.8. SURDEFINITION (SURCHAGE) DE METHODES :

6.8.1. : Définition :

La surdéfinition (surcharge) d'un symbole conduit à plusieurs significations différentes du symbole entre lesquelles on choisit en fonction du contexte. Le symbole peut être un opérateur +(addition, concaténation, etc. …) mais également en JAVA des méthodes.

Les méthodes usuelles et statiques peuvent être redéfinies.
Une méthode publique ou privée peut être redéfinie.

Plusieurs méthodes peuvent avoir le même nom à condition que le nombre et le type de leurs arguments permettent au compilateur d'effectuer un choix, ie à condition qu'elles aient des **signatures différentes** (polymorphismes).

La signature d'une méthode est constituée du nom de la méthode suivi de la liste ordonnée des types de ses arguments. Le nom des arguments et le type de retour ne sont pas considérés dans la signature.
Les méthodes avec même signature mais de types de retour différents sont interdites.

6.9. : TRANSMISSION D'INFORMATIONS AVEC LES METHODES :

6.9.1. : Transmission par valeur :

La transmission d'un argument à une méthode et la transmission du résultat d'une méthode se fait par *valeur* : la méthode reçoit une copie de la valeur de l'argument effectif.

Cependant, avec une variable de type objet, la méthode reçoit une copie de la référence.
Ainsi, la méthode peut modifier l'objet initial qui n'a pas été recopié par le biais de ses champs ou de ses méthodes. La méthode appelée a accès à l'objet par sa référence.
Une modification de la référence à l'intérieur de la méthode appelée, par exemple une nouvelle instance, n'est pas transmise à la méthode appelante par ses arguments, il faut utiliser sa valeur de retour.

En conclusion :
-la transmission des types primitifs se fait par valeur.
-la transmission des objets se fait par référence (adresse).

Ainsi, une méthode ne peut pas modifier la valeur d'un argument effectif d'un type primitif (après l'appel d'une méthode).

Moyens pour transmettre la valeur d'une variable de type primitif à l'extérieur de la méthode appelée :

-retourner la variable pour la méthode elle-même.
-créer un objet et le passer en argument car les méthodes peuvent opérer sur les champs directement ou par le biais de méthodes d'accès.

6.9.2. : Transmission d'objet en argument :

Il est possible d'utiliser des arguments de type classe.

Exécution:
Racine1=-0,21
Racine2=-4,79
Racine1=-0,21
Racine2=-4,79
Nombre de systèmes=2
Facteur multiplicatif des coefficients des deux systèmes=2.0

->Créer une méthode statique qui va tester s'il y a un facteur multiplicatif entre les coefficients des deux équations.

```
Import java.txt.DecimalFormat ;
Class Equation
```

```
{
…
//Méthode de test des coefficients dans deux systèmes.
//Méthode de classe (indépendante des objets)
public static double tesCoefficients(Equation E1, Equation E2)
{
final double eps=1E-10 ;
double facteur=0.0;
facteur=E2.coeffX2/E1.coeffX2;
        if ((((E2.coeffX1/E1.coeffX1)-facteur)>eps)||
            (E2.coeffX0/E1.coeffX0)-facteur)>eps) facteur=0 .0 ;
return facteur ;
}
…
}


//classe test
public class EquationSecondDegre
{
public…
{
equation uneEquation= new Equation (1.0;5.0,1.0);
uneEquation.Résolution();
uneEquation.affichage();
…..
Equation uneEquation1 = new Equation (2.0,10.0,2.0) ;
UneEquation1.Résolution();
UneEquation1.affichage():
….
System.out.println(« Nombre de systèmes= »+Equation.nbSystème()) ;
…..
}
}
```

Une méthode d'un objet ou d'une classe t (méthode appelée pour un objet o) recevant en argument un objet p de la même classe t, a accès aux champs et méthodes privés de p : *l'unité d 'encapsulation est la classe* et non l'objet.
L'autorisation d'accès concerne tous les objets de la classe et non, seulement, l'objet courant.

Une méthode d'une classe A recevant en argument un objet de classe B?A, n'a pas accès aux champs et méthodes privés de B.

6.9.3. :Transmission par valeur de types primitifs :

1ère version :

```
class Equation
{
…
```

```
//Méthode de doublement des coeffs ax^2+bx+c (ne marche pas)
public void doublementCoeffs(double a, double b, double c)
{
a=2*a ;
b=2*b ;
c=2*c;
….


//Méthode principale
…
double cX2=1.0;
double cX1=5.0;
double cX0=1.0;
…
uneEquation.doublementCoeffs(cX2,cX1,cX0) ;
//Transmission par valeur cX2, cX1, cX0 n'ont pas leurs valeurs doublées.
```

Exécution :

```
Equation=1.00*2+5.00+1.00
Racine1=-0,21
Racine2=-4.79
Equation=1.00*2+5.00+1.00
…
```

2ème version :

```
….
a=2*a ;
….
System.out.println(« a= »+a+ »b= »+b+ »c= »+c) ;
Retun a;
}
…
}
```

6.9.4.: Transmission par adresse de la référence d'un objet :

1ère version avec un seul objet modifié

```
class Equation
{
…
public void doublementCoeffs()
{
coeffX2=2*coeffX2;
coeffX1=2*coeffX1;
coeffX0=2*coeffX0;
}
…
```

}

Dans le main : uneEquation.doublmentCoeff() ;

2ème version avec deux objets :

```
class Equation
{
…
public Equation doublementCoeff()
{
Equation E= new Equation();
E.coeffX2=2*coeffX2;
E.coeffX1=2*coeffX1;
E.coeffX0=2*coeffX0;

Return E ;
}
…
}
```

Dans le main : Equation uneEquation1 = uneEquation.doublementCoeff() ;

6.9.5. : Valeur de retour d'une méthode :

L'instruction return est suivie d'une expression de même type que celui de la méthode. De plus, elle termine l'exécution d'une méthode et provoque un retour à la méthode d'appel (méthode appelante).

Il est possible d'avoir plusieurs return dans une méthode, par exemple en cas d'instructions conditionnelles . De préférence, un seul return.

Le retour d'une méthode se fait par copie :

- -par valeur pour un type primitif.
- -par adresse pour un objet (copie de la référence).

6.9.6. :Autoréférence this :

3 propriétés :

(1) Le mot-clé this est une référence à l'objet courant dans sa globalité.

(2) this remplace le nom de l'instance courante.
Exemple :E.coeffX2=2*this.coeffX2 ;

(3) utilisation de noms d'arguments identiques à des noms de champs.
Cette utilisation permet d'éviter de créer de nouveau identificateurs.
Exemple :this.coeffX2=coeffX2 ;

(4) appel d'un constructeur au sein d'un autre constructeur :

Un constructeur peut appeler un autre constructeur de la même clase (et portant alors sur l'objet courant), même le constructeur vide mais pas le constructeur implicite qui disparaît dès qu'un constructeur est créé.

Ce constructeur est appelé avec un nom de méthode this(), donc suivi de parenthèses entourant les arguments dans la première instruction du constructeur.

This() est remplacé par le nom de la classe, donc par un constructeur.

Simplification du programme précédent en utilisant l'appel d'un constructeur dans un constructeur :

```
Class Equation
{
…
//Constructeur vide
public Equation()
{
this(0.0,0.0,0.0) ;
}
//Constructeur à 3 arguments pour ax2+bx+c
public Equation(double X2, double X1, double X0)
{
coeffX2=X2 ;
coeffX1=X1 ;
coeffX0=X0 ;
nbresolution++ ;
}
//Constructeur à 2 arguments pour ax2+c
public Equation (double X2, double X1, double X0)
{
this(X2,0.0,X0) ;
}
……..
}
……….
```

6.10. : RECURSIVITE DES METHODES :

2 types de récursivité :

-récursivité directe : une méthode comporte au moins un appel à elle-même

-récursivité croisée : l'appel d'une méthode entraîne l'appel d'une autre méthode qui à son tour appelle la méthode initiale.

Possibilité de cycles faisant intervenir plus de 2 méthodes.

La récursivité s'applique aux méthodes usuelles et aux méthodes statiques(de classe).

Programme Factoriel :

```
Package coursjava ;

Class util
{
        public static long fac(long n)
                {
                if (n>1) return (fac(n-1)*n);
                else return 1;
                }
}

public class Factoriel
{
        public static void main(String[ ] args)
        {
int n;

        System.out.print("Donner un entier positif");
        n=Lecture.lireInt() ;
        System.out.println(« Résultat= « +util.fac(n)) ;
        }
}
```

Exécution :

Donner un entier positif : 10
Résultat=3628880

/*NON ETUDIE CETTE ANNEE*/

6.11. : CLASSES INTERNES :

Les classes internes sont principalement, mais non exclusivement, utilisées en programmation évènementielle.

Une classe est interne lorsque sa définition est située à l'intérieur d'une autre classe.

```
class Externe
{
class Interne
{
//champs et méthodes de la classe interne
}
//champs et méthodes de la classe interne
}
```

Utilisation de la classe interne dans une méthode de la classe externe :

```
Class Externe
{
class interne
{
//Champs et méthodes de la classe interne
}
//champs et méthodes de la classe externe
//méthode de la classe externe

public void methodeExterne()
{
interne i= new interne();
}
}
```

Utilisation de la classe interne comme champ de la classe externe:

```
Class externe
{
class interne
{
//champs et méthodes de la classe interne

//champ de la classe externe
private interne i ;
………
}
```

Liens entre objet interne et objet externe :

(1) *Un objet d'une classe interne est toujours associé au moment de son instanciation à un objet d'une classe interne qui lui a donné naissance.*

(2) Un objet d'une classe interne a toujours accès aux champs et aux méthodes (même privés) de l'objet externe lui ayant donné naissance.
(3) Un objet d'une classe externe a toujours accès aux champs et aux méthodes (même privés) de l'objet interne auquel il a donné naissance.

Une méthode statique d'une classe externe ne peut créer aucun objet d'une classe interne car une méthode statique n'est associée à aucun objet.
Une classe interne ne peut pas contenir des champs statiques et des méthodes statiques.

Utilisation de la classe interne en dehors de la classe externe.
Référence à un objet de type interne

Externe Interne i

Création d'un objet de type interne :

Externe e= new Externe()
I = new e.Interne()
Ou
Externe.Interne I= new Externe.Interne()
// syntaxe particulière de new.

Utilisation d'une classe interne dans une méthode de la classe externe :
L'instanciation d'objet de type interne ne peut se faire que dans la méthode.

Utilisation de classe interne statique :
La classe interne n'a plus accès aux champs et méthodes de la classe externe sauf aux champs statiques et méthodes statiques.

6.12. : PACKAGES :

6.12.1. : Définition :

Un package est un regroupement logique sous un identificateur commun d'un ensemble de classes (bibliothèque).
Le package permet un classement hiérarchique des classes pour :
-ranger les classes selon des fonctionnalités communes.
-sécuriser l'utilisation des classes :le risque de créer 2 classes de même nom se trouve limité aux seules classes d'un même package. Deux classes de même nom peuvent être dans 2 packages différents.

Un package contient des classes et des sous-packages.
Le nom du package est en relation avec le nom du répertoire qui le contient, les niveaux supérieurs pouvant être quelconques.

Exemples :

-le package coursjava est placé dans un répertoire coursjava, par exemple D:\…\coursjava
-le sous-package chapitre1 du coursjava est placé dans un répertoire D:\…\coursjava\chapitre1

6.12.2. :Attribution d'une classe à un package :

Un package est caractérisé par un nom qui est :
-soit un identificateur, par exemple coursjava
-soit une suite d'identificateurs séparés par des points, par exemple *coursjava.chapitre1*

L'attribution d'une classe à un package se fait dans la 1ère ligne du fichier source

package nom_package

Exemples :
Package coursjava
Package coursjava.chapitre1

Par convention, les noms de package sont en lettres minuscules.
La hiérarchie des packages se fait du général au particulier.

Toutes les classes d'un même fichier source appartiennent donc toujours à un même package.
Si aucun package n'est déclaré, les classes du fichier source appartiennent au package par défaut.

6.12.3. :Utilisation d'une classe d'un package :

Lors de la référence d'une classe dans un programme, le compilateur recherche la classe dans le package par défaut.
Pour utiliser une classe appartenant à un autre package, il existe 3 possibilités :

(1)Donner le nom du package avec le nom de la classe.

nom_package.Nom_Classe

```
//classe test
public class EquationSecondDegre
{
//méthode principale
public static void main (String [ ] args)
{
coursjava.Equation uneEquation= new coursjava.Equation(1.0,5.0,1.0);
uneEquation.résolution();
```

```
uneEquation.affichage();
….
}
}
```

Il est inutile de préciser le nom du package pour les champs et les méthodes après la création de l'objet.

(2) Importer la classe

import nom_package .Nom_Classe

en début du fichier source.

```
Import coursjava.Equation;
//classe Test
public class EquationSecondDegre
{
//méthode principale
public static void main (String [ ] args)
{
Equation uneEquation = new Equation(1.0,5.0,1.0);
…
}
}
```

(3) Importer le package

*import nom_package.**

en début du fichier source

```
import coursjava.Equation;
//classe Test
import coursjava.*;

public class EquationSecondDegre
{
//méthode principale
public static void main (String [ ] args)
{
Equation uneEquation = new Equation(1.0,5.0,1.0);
…
}
}
```

<u>Remarques:</u>

(1) "import coursjava.*" (sans le caractère de substitution*) n'importe que les classes de coursjava et pas les classes des sous-packages éventuels.

(2) il n'est pas possible d'utiliser le caractère de substitution * pour des packages (uniquement pour des classes).

(3) Le compilateur ne compile que les classes utilisées dans un package comportant également des classes inutilisées.

6.12.4. : Packages standards :

Packages standards : java.lang, java.awt, java.awt.event, java.swing, etc., …
Le package standard java.lang est automatiquement importé à la compilation permettant d'avoir des classes standards : Math, System, Integer, Double, etc., …

6.12.5. : Portée des classes :

La portée d'une classe est définie comme la zone dans laquelle la classe est accessible.
Cette portée est déterminée par l'attribut de portée lors de la déclaration de la classe :
-avec l'attribut public : la classe est accessible à toutes les autres classes.
-avec l'absence d'attribut, la classe n'est accessible qu'aux classes du même package.
Les classes non publiques sont généralement des classes auxiliaires du package.

L'attribut public n'a pas d'importance quand les classes du projet utilisant le package par défaut, à l'exception de la classe contenant main qui doit être publique pour pouvoir être reconnue par la machine virtuelle.

6.12.6. : Portée des champs et des méthodes :

-avec l'attribut public :
les champs et les méthodes sont accessibles depuis l'intérieur de la classe.
-avec l'attribut private :
les champs et les méthodes ne sont accessibles qu'aux méthodes de la classe.
-avec l'absence d'attribut :
les champs et les méthodes ne sont accessibles qu'aux classes du même package (accès de package).

A noter :
-avec l'attribut protected :
les champs et les méthodes sont accessibles aux classes du même package ainsi qu'à ses classes dérivées qu'elles appartiennent ou non au même package.

VII. TABLEAUX

7.1. :DECLARATION ET CREATION DE TABLEAUX :

7.1.1 Introduction :

Un tableau est un ensemble d'éléments de mêmes types, désigné par un nom unique. Chaque élément étant obtenu par un indice de position sur cet ensemble.
Un tableau est un objet.

7.1.2. : Déclaration d'un tableau :

La déclaration consiste à donner la référence à u, tableau.

La référence à un tableau précise le type des éléments du tableau, sans donner de dimension au tableau.(Ceci se fera à la création)
La valeur de cette référence est nulle.

Plusieurs possibilités de déclaration :

int t[ ] ou int[ ] t

int[ ] t1, t2 est équivalent à int t1[ ], t2[ ]

Les éléments d'un tableau peuvent être de type primitif ou de type objet.

Exemple : Equation t[ ]

Remarque :

Int t[10] est rejeté à la compilation :
Interdiction de donner une dimension.

7.1.3. : CREATION D'UN TABLEAU :

i) Création par l'opérateur new.

La valeur de l'expression fournie à l'opérateur *new* n'est calculée qu 'au moment de l'exécution (non fixée à la compilation)

System.out.println(« Donner la dimension du tableau : ») ;
Int n = Lecture.lireInt() ;

```
Int t[ ] = new int [n] ;
```

Ou

```
Int t[ ];
T= new int [n];
```

Ou

```
Final int  DIM_TAB=5;
int[ ]t= new int [DIM_TAB];
```

La dimension du tableau peut varier d'une exécution à l'autre. Une fois l'objet tableau créé, sa taille ne peut plus être modifiée mais sa référence peut être modifiée en désignant d'autres tableaux.

ii) Création par initialisation :

```
Int t[ ] = {1,2,3,4,5} ;
```

Création d'un tableau de 5 entiers de valeurs respectives 1, 2, 3, 4, 5 et placement de la référence dans t.

```
Double t[ ] = {1.1, 2.2, 3.3 } ;
```

Création d'un tableau de trois réels.

7.2. : UTILISATION DE TABLEAUX :

7.2.1. : Accès individuel aux éléments du tableau :

Le premier élément d'un tableau a obligatoirement l'indice 0.
Le dernier élément a pour indice la dimension du tableau diminuée de 1.

On accède à un élément du tableau en suffixant son nom par l'indice entre crochets. L'indice peut être une expression entière.

T[i]

Quand la valeur de l'indice dépasse l'intervalle de la taille du tableau, une exception du type ArrayOutOfBoundsException est déclenchée (levée).

Accès aux éléments du tableau :

```
System.out.println(« Donner la dimension du tableau : ») ;
Int n= Lecture.lireInt() ;
Int t[ ] = new int [n] ;
For (int I=0; I<n;I++) ……=t[I];
```

7.2.2. : Accès global au tableau (affectation de références) :

L'affectation t1=t2 recopie dans t1 la référence contenue dans t2.
Les références t1 et t2 désignent le même tableau qui était initialement associé à t2.
La modification d'un élément de ce tableau par la référence t1 se retrouvera avec la référence t2.
Le tableau anciennement référencé par t1 est candidat au ramasse-miettes.

L'affectation de références de tableaux n'entraîne aucune recopie des valeurs des éléments au tableau, même situation avec l'affectation des objets.

7.2.3. : Taille d'un tableau :

Le champ length permet de connaître le nombre d 'éléments d'un tableau de référence qui évolue au cours de l'exécution du programme.
Length n'est pas une méthode.
Aucun champ ne devrait être public. Mais il est impossible de le modifier parce qu'il est déclaré avec le modificateur final.

Exemple :

```
For (int i=0 ;i<t.length;I++)
```

7.3.: TABLEAU D'OBJETS:

Programme avec tableau :

```
//classe test
public class EquationSecondDegre
{
//Méthode principale
public static void main (String [ ] args)
```

```
{
Equation systeme [ ] = new Equation [2];

Système [0]= new Equation (1.0,5.0,1.0);
Système [1]= new Equation (1.0,10.0,1.0);



For (int i=0 ;i<systeme.length;i++)
{
systeme[i].résolution();
systeme[i].affichage();
System.out.println("……");
}
System.out.println("Nombre de systèmes="+Equation.nbSysteme());
}
}
```

7.4. : TABLEAU EN ARGUMENT :

Comme avec les objets, lorsqu'on transmet un nom de tableau en argument d'une méthode, on transmet une copie de la référence au tableau.
La méthode agit alors directement sur le tableau (transmission par adresse).

Programme tableau en argument :

```
Package coursjava ;
Import java.text.DecimalFormat ;

//classe Equation
{
…

//Méthode de doublement des coeffs aX^2+bX+c
public static void doublementCoeffs(double Coeff[ ])
{
for (int i=0 ;i<systeme.length;i++)
{
coeff[i]=2*Coeff[i];
}
…
}
```

```
//class Test
public class EquationSecondDegre
{
//Méthode principale
public static void main (String[ ] args)
{
double coeff[ ]={1.0,5.0,1.0};
Equation systeme[ ]= new Equation[2];

Systeme[0]=new Equation(coeff[0],coeff[1],coeff[2]);
Equation.doublementCoeff(coeff);
Systeme[1]= new Equation (coeff[0]….);


for (int i=0 ;i<systeme.length;i++)
{
systeme[i].résolution();
systeme[i].affichage();
System.out.println("……");
}
System.out.println("Nombre de systèmes="+Equation.nbSysteme());
}
}
```

7.5.: TABLEAUX MULTIDIMENSIONNELS:

Java ne possède pas la notion de tableaux à plusieurs indices.
Il simule cette notion avec des tableaux de tableaux, ie des tableaux dont les éléments sont des tableaux. Cette possibilité permet de créer des tableaux irréguliers .
Par exemple : une matrice triangulaire.

Déclaration d'une référence t à un tableau à deux dimensions

Int t[ ] [ ]

Ou

Int [ ] t[ ]

Ou

Int [ ] [ ] t

t est une référence à un tableau dans lequel chaque élément est lui-même une référence à un tableau d'entiers.

Création d'un tableau à deux dimensions :

Final int DIM1=3
Final int DIM2=4
t= {new int[DIM1], new int [DIM2]}

T[0] est la référence au 1er tableau de trois entiers.
T[1] est la référence au 2ème tableau de 4 entiers.

T[0] [0] est le 1er entier du 1er tableau.
T[1] [0] est le 1er entier du 2ème tableau.

Un tableau a un champ length qui est égal à sa dimension s'il est mono dimensionnel et à sa première dimension s'il est multidimensionnel.

t.length=2
t[0].length=3
t[1].length=4

Exemple : Création d'une matrice triangulaire:

```
Int t[ ] [ ] = new int [NLIG] [ ]
For (int I=0; I<T.length;I++)
{
t[I]= new int [I+1];
```

Initialisation d'un tableau multidimensionnel:

```
Int t[ ] [ ] = { { 1,2,3},{1,2,3,4}} ;
```

Accès aux éléments d'un tableau multidimensionnel :

```
For (int I=0; I<T.length;I++)
        {
        For (int j=0; j<t[I].length;j++)
                {
                ……
                t[I] [j];
                }
        }
```

Déclaration et création d'un tableau régulier à deux dimensions:

```
Int t[ ] [ ] = new int [NLIG] [NCOL]

Ou

Int t[ ];
T=new int [NLIG] [NCOL]

Ou

Int t[ ] [ ] = new int [NLIG] [ ] ;
For (int I=0; I<length;I++)
        {
        t[I]= new int [NCOL];
        }
```

VIII. HERITAGE:

8.1.: INTRODUCTION:

L'héritage permet d'utiliser des classes, rendant ainsi le développement des programmes plus rapide, sûr et lisible.
Il permet de définir une nouvelle classe dite *classe dérivée* (sous-classe ou classe descendante), à partir d'une classe existante dite *classe de base* (super-classe ou classe ascendante).
*/*Cette classe dérivée hérite des champs et des méthodes privés et publics.*/*
Elle ne peut modifier que les champs et méthodes publics.
Elle peut posséder de nouveaux champs et méthodes.

Le mot-clé extends permet de définir une sous-classe.

8.2. ACCES D'UNE CLASSE DERIVEE AUX MEMBRES DE SA CLASSE DE BASE :

Les membres (champs et méthodes) publics de la classe de base restent des membres publics de la classe dérivée. Une méthode d'une classe dérivée a accès aux membres publics de sa classe de base.

Une méthode d'une classe dérivée n'a pas accès aux membres privés de sa classe de base (principe d'encapsulation).

Possibilité d'avoir une classe de base sans constructeur et avec une classe dérivée sans/avec constructeur.

8.3. :CONSTRUCTION ET INITIALISATION DES OBJETS DERIVES :

8.3.1. : Appel du constructeur :

(i) Rappel de l'appel du constructeur dans le cas d'une classe simple (non dérivée)

-création d'un objet par new sans argument : appel du pseudo-constructeur par défaut (la classe ne comporte aucun constructeur)
-création d'un objet par new avec argument : appel du constructeur ayant la même signature (nombre et type des arguments).

(ii) Cas d'une classe de base avec un constructeur et une classe dérivée avec un constructeur :

Pour initialiser des champs de la classe de base encapsulés (attibut private), la classe dérivée doit posséder des méthodes d'altération ou utiliser le constructeur de la classe de base.

Le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet.

Un constructeur d'une classe dérivée appelle un constructeur d'une classe de base avec sa 1ère instruction.
Le constructeur appelé est désigné par le *mot-clé super* suivi éventuellement de ses arguments entre parenthèses.

```
//classe dérivée
class ClasseDerivee extends Equation
{
public ClasseDerivee(double a, double b, double c)
{
super(a,b,c) ;
}
}
```

Le *mot-clé this* permet d'appeler dans un autre constructeur de la même classe. Comme *this* et *super* doivent correspondre à la 1ère instruction du constructeur, il n'est pas possible d'utiliser simultanément *this et super*.

L'appel par *super* ne concerne que le constructeur de la classe de base de niveau immédiatement supérieur.

Classe de base sans constructeur :

```
Class A
{
//sans constructeur
…..
}

class B  extends A
{
public B()
{
super();
….
}
}
```

L'appel super() est non obligatoire.

Classe dérivée sans constructeur
Il y a appel du pseudo-constructeur par défaut sans argument.
Dans le cas d'une classe simple, ce pseudo-constructeur ne fait rien.
Dans le cas d'une classe dérivée, ce pseudo-constructeur appelles un constructeur sans argument de la classe de base.

Soit le pseudo-constructeur par défaut sans argument de la classe de base :

```
Class A
{
//Pas de constructeur
…..
}
class B
{ // Pas de constructeur
}
```

La construction d'un objet de type B, B b=new B(), entraîne l'appel du constructeur par défaut de A.

Soit le constructeur public sans argument de la classe de base :

```
Class A
{
// constructeur1
```

```
public A()
{…}

//constructeur2
public A(int n)
{….}
……
}

class B extends A
{
// pas de constructeur
}
```

Le constructeur d'un objet de type B, B b=new B() entraîne l'appel du constructeur sans argument de A.

```
Class A
{
//constructeur2
public A (int n)
{..}
…..
}

class B extends A
{
// pas de constructeur
}
```

La construction d'un objet de type B entraîne une erreur de compilation car le consructeur par défaut cherche à appeler un constructeur sans argument de A qui n'existe pas. Il ne peut également pas appeler le constructeur par défaut de A puisqu'il existe un constructeur avec argument.

De préférence, définir systématiquement un constructeur vide dans chaque classe.

8.3.2. : Initialisation d'un objet dérivé :

Rappel sur les phases de création d'un objet de classe simple :

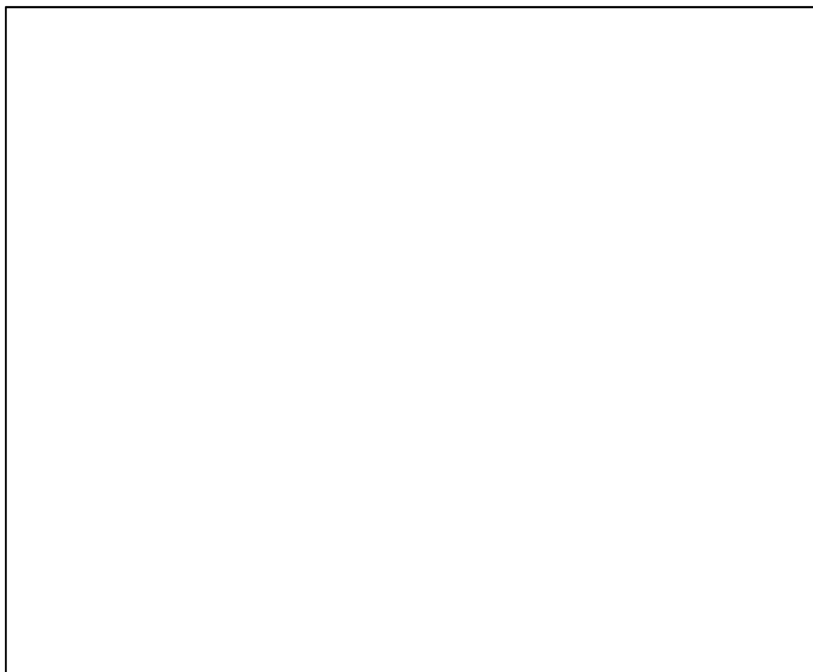
-allocation mémoire
-initialisation par défaut des champs
-initialisation explicite des champs
-exécution des instructions du constructeur.

Phases de création d'un objet de classe dérivée avec class B extends class A :

-allocation mémoire de l'objet de type B : champs hérités et champs propres.
-initialisation par défaut de tous les champs (hérités de A et propres à B) de B aux valeurs « nulles ».
-initialisation explicite des champs hérités.
-exécution des instructions du constructeur de A
-initialisation explicite des champs propres
-exécution des instructions du constructeur de B

8.4. : DERIVATION SUCCESSIVES :

Chaque classe ne peut avoir qu'une *super-classe* (pas d'héritage multiple) mais peut avoir plusieurs sous-classes.



```
Class Point
{
private int x,y ;

public Point(int x, int y)
{
this.x=x ;
y=y;
}

public Point()
{
this(0.0);
}
…
}

class Cercle extends Point
{
…….;
}
```

```
public class Test
{
```

```
public static void main(String [ ] args)
{
Point p,p1;
Cercle c, c1 ;

P=new Point(10,20) ;
C=new Cercle(30,40,50);

P1=c;    //affecte la réf. De la classe dérivée Cercle à la réf. De la super-classe Point

C1=(Cercle)p1 ;  // remplacement du type de la réf. De la super classe Point par la réf. De
la  classe dérivée Cercle.

C1=(Cercle)p ;   //Erreur d'exécution.

If (p instanceof Cercle) c1=(Cercle) p ;
Else
System.out.println(« p ne référence pas un cercle ») ;
}
```

8.5. :REDEFINITION ET SURDEFINITION DE MEMBRES :

8.5.1. : Redéfinition de méthodes :

Surdéfinition de méthodes à l'intérieur de la même classe= méthodes de même nom mais de signatures différentes.
La surdéfinition permet de cumuler plusieurs méthodes de même nom.

Redéfinition de méthodes dans une classe dérivée = méthodes de même nom (surdéfinition), de même signature et de même type de valeur de retour.
La redéfinition permet de substituer une méthode par une autre.

Une classe dérivée permet de redéfinir une méthode de sa classe de base ou plus généralement d'une classe ascendante, en masquant la méthode de la classe de base.
La redéfinition d'une méthode s'applique à une classe et à toutes ses descendantes jusqu'à ce que l'une d'entre elles redéfinisse éventuellement, à nouveau, la méthode.

Exemple :

Pour appeler la méthode affichage de la classe de base, il faut préfixer affichage par le mot *super*.

8.5.2. :Surdéfinition (surcharge) de méthodes :

Une classe dérivée peut surdéfinir une méthode de sa classe de base ou plus généralement d'une classe ascendante.

```
Class A
{
public void m (int n)
{
…..
}

……….
}

class B extends A
{
public void m(double x)
{
…….
}
……
}
```

```
A a; B b; int n; double x;
a.m(n); appel de m(int n)
a.m(x) ; arreur de compilation
b.m(n) ; appel de m(int n)
b.m(x) ;appel de m(double x).
```

8.5.3. : Utilisation simultanée de redéfinition et de surdéfinition :

Peut conduire à des résultats complexes.

8.5.4. :Contraintes portant sur la redéfinition :

Valeur de retour :

Lors de la surdéfinition d'une méthode, il n'est pas nécessaire de respecter le type de la valeur de retour.
Lors de la redéfinition d'une méthode, il faut respecter le type de la valeur de retour.

Droits d'accès :

La redéfinition d'une méthode ne doit pas diminuer les droits d'accès à cette méthode.
Elle peut les augmenter, ie étendre sa zone de lisibilité.

8.5.5. : Règles générales de redéfinition et de surdéfinition :

Règles de redéfinition d'une méthode d'une classe ascendante par une classe dérivée.

-même signature.
-valeurs de retour des deux méthodes de même type.
-droits d'accès de la méthode de la classe dérivée au moins supérieure………
………………

8.5.6. :Duplication des champs :

Une classe dérivée peut définir un champ portant le même nom qu'un champ d'une classe ascendante.

8.6. : POLYMORPHISME :

8.6.1. : Définition :

Le polymorphisme permet de manipuler des objets sans en connaître totalement leur type dans un contexte d'héritage.(de classes dérivées)
Chaque objet réagit en fonction de son propre type.
A une variable objet peut être affectée une référence à un objet du type correspondant mais également une référence à un objet de type dérivé.

conversion implicite d'une référence à un type classe T en une référence à un type ascendant de T.

Le choix d'une méthode lors de l'exécution s'appelle ligature dynamique (liaison dynamique).

Le polymorphisme se généralise à une hiérarchie de plusieurs classes.

8.6.2. :Polymorphisme et gestion d'un tableau hétérogène :

Le polymorphisme permet de gérer un tableau hétérogène, ie dans lequel les éléments peuvent être de types différents.

```
    Package coursjava ;
Class Equation
{
...
public void affichage()
{
…..
}
}

//Classe dérivée
class ClasseDérivée extends Equation.
{
…..
//Méthode d'affichage
public void affichage()
…..

//Classe Test
public static void main (String[ ] args)
{
Equation système[ ]=new Equation [2];

Systeme[0]=new Equation(1.0,5.0,1.0);
Systeme[1]=new ClasseDerivee(2.0,10.0,2.0);

For(i==0;i<systeme.legth;i++)
{
systeme[i].résolution;
systeme[i].affichage;
System.out.println("...");
```

Execution:

Equation=1.00*x2+5.00x+1.OO

Racine1=-0,21

Racine2=-4,79

8.6.3. : Polymorphisme et absence de méthode dans une classe dérivée.

```
Package coursjava ;
Class Equation
{
...
public void affichage()
{…
}
}

class ClasseDerivee extends  Equation
{
//public void affichage()
//{
//System .out.println(" Equation 2nd degre");
//super.affichage();
//}
}
```

Pour classe Test pareil qu'avant

8.6.4. : Polymorphisme et structuration des objets :

Le polymorphisme permet de structurer les objets avec des méthodes communes au niveau des classes ascendantes et des méthodes spécifiques au niveau de toutes les classes (ascendantes et descendantes).

Les méthodes communes qui ne sont pas redéfinies dans les classes descendantes, appellent les méthodes spécifiques.

Mais, seule la méthode spécifique de la classe correspondant à l'objet est effectivement appelée (celui de référence this).

```
Package coursjava ;
Import java.text ;DecimalFormat ;

//Classe Equation
class Equation
```

```
{
…
public void affichageCommun()
{
affichageSpécifique;
...
}
public void affichageSfécifique()
{
System.out.println(« Equation du 2nd degré ») ;
}
}
class ClasseDérivée extends Equation
{
public ClasseDérivée (double a, double b, double c)
{
super(a,b,c) ;
}

public void affichageSpécifique()
{
System.out.println(« Equation du second degré de la classe dérivée ») ;
….
}
```

8.6.5. : Polymorphisme et surdéfinition :

Le polymorphisme se fonde classiquement sur la redéfinition de méthodes. Mais, il peut également se baser sur la modification de méthodes pouvant conduire à des situations complexes.

/*NE PAS APPRENDRE

8.6.6. : Règles du polymorphisme :

Compatibilité :

Il existe une conversion d'une référence à un objet de classe T en une référence à un objet d'une classe ascendante de T. Cette conversion intervient aussi bien dans les affectations que les arguments effectifs.

Ligature dynamique :

Dans un appel de la forme x.f(…) où x est supposé de classe T, le choix de f est déterminé de la façon suivante :

-A la compilation :

On détermine dans la classe T ou ses ascendantes, la signature de la meilleure méthode f convenant à l'appel, définissant ainsi le type de la valeur de retour.

-A l'exécution :
On recherche la méthode f de signature et de type de retour voulus, à partir de la classe correspondant au type effectif de l'objet, référencé par x. Cet objet est obligatoirement de type T ou descendant.
Si cette classe ne comporte pas de méthode appropriée, on remonte dans la hiérarchie jusqu'à ce que l'on en trouve une, à la limite on remonte jusqu'à T.
*/

8.6.7. : Opérateur instanceof :

L'opérateur binaire instanceof prend un objet comme 1er opérande et une classe comme 2e opérande.
Le résultat vaut true si l'objet est une instance de la classe et false dans le cas contraire.

Exemple :
Equation p =new Equation(1.0,5.0,1.0);
If (p instanceof Equation) System.out.println("OK");

8.6.8.: Mot-clé super:

Le mot-clé super permet d'accéder à une méthode d'une classe de base.

8.7. : SUPER CLASSE OBJET :

8.7.1. : Définition :

Il existe 3 types de classe :

-les classes simples :
-les classes dérivées
-la super classe Objet dont dérive implicitement toutes les classes simples.

```
Class Equation
{
…
}
```

est équivalent à :

```
class Equation extends Objet
{
….
}
```

8.7.2.: Utilisation d'une référence de type Object:

Une variable de type Object peut être utilisée pour référencer un objet de type quelconque.

```
Equation uneEquation =new Equation(1.0,5.0,1.0) ;
Object o;

o=uneEquation;
```

Cette particularité peut être utilisée pour manipuler des objets dont on ne connaît pas le type exact à un certain moment.

Cependant, pour appliquer une méthode précise à un objet référencé par une variable de type Object, il faut effectuer une conversion puisque la méthode n'existe pas dans la classe Object.

```
Equation uneEquation =new Equation(1.0,5.0,1.0) ;
Object 0;

0=uneEquation;

((Equation)0).affichage() ;

//OU

Equation uneAutreEquation = (Equation)0 ;
UneAutreEquation.affichage() ;
```

8.7.3. : Utilisation de la méthode toString de la classe Object :

La méthode toString fournit un objet de type String avec une chaîne de caractères contenant :

-le nom de la classe concernée (correspondant à l'objet référencé) même si la méthode toString n'a pas été redéfinie.
-l'adresse de l'objet en hexadécimal précédée de @.

```
//Classe Test
public class EquationSecondDegre
{
//Méthode principale
public static void main(String[ ] args)
```

```
{
Equation systeme[ ]= new Equation[2];

Systeme[0]=new Equation(1.0,5.0,1.0);
Systeme[1]=new ClasseDérivée(2.0,10.0,2.0);

System.out.println(“Sysqteme[0]=”+Systeme[0].toString());
System.out.println(“Sysqteme[1]=”+Systeme[1].toString());
….
```

Exécution :
Systeme[0]=coursjava.Equation@310d42
Systeme[1]=coursjava.ClasseDérivée@5d87b2

Il est possible de redéfinir la méthode toString dans une classe donnée.

La méthode toString d’une classe possède également la propriété d’être automatiquement appelée en cas d’une conversion implicite en chaîne avec l’opérateur+ comportant un opérande de type chaîne.
Cette conversion est indépendante du type de la référence o et du type de l’objet effectivement référencé par o.

```
Class ClasseDérivée extends Equation
{
…..
public String toString()
{
return “Redéfinition de la méthode toString”;
}
……
```

8.7.4. : Utilisation de la méthode equals de la classe Object :

La méthode equals compare les adresses de 2 objets : 01equals(02)
Il est possible de redéfinir la méthode equals dans une classe donnée.

```
System.out.println(« Egalité des adresses= »+systeme[0].equals(systeme[1]) ;
```

```
Class Equation
{
…..
public boolean equals(classeDérivée e)
{
return ((coeffX2==e.coeffX2)
&& (coeffX1==e.coeffX1)
```

```
&& (coeffX0==e.coeffX0);
}
}
......
```

8.7.5. : autres méthodes de la classe Object :

protected Object …………….
protected void finalize() …………
public final class getclass()

8.7.6.: Tableaux et classe Object:

Les tableaux ne possèdent qu'une partie des propriétés des objets.
Un tableau peut être considéré comme appartenant à une classe dérivée de Object.

```
Object o ;

o=new int[5] ;
```

Le polymorphisme s'applique aux tableaux :
Si la classe B dérive de la classe A, un tableau de B est compatible avec un tableau de A.

```
Class B extends A
{
……
}

A=tA[ ];
B=tB[ ];

tA=tB ;  //La réciproque est fausse
```

Il est impossible de dériver une classe d'une hypothétique classe tableau

```
Class impossible extends int[ ]
```

8.8.: CLASSES ET METHODES FINALES:

Rappel :

Le *mot-clé final* appliqué à des variables locales ou à des champs d'une classe, interdit la modification de leur valeur.

Une méthode déclarée *final* ne peut pas être redéfinie dans une classe.
Une classe déclarée *final* ne peut plus être dérivée.

Classe non finale + méthode finale # classe finale

Avantages :
-détection d'erreur à la compilation et non à l'exécution.
-optimisation du code

Inconvénients :
-impossibilité de modification du code.

8.9. :CLASSES ABSTRAITES :

8.9.1. : Définition :

Une classe abstraite est une classe qui ne permet par d'instancier des objets. Elle sert de classe de base pour une dérivation

```
Abstract class A
{
…..
}
```

Une classe abstraite comporte des champs et des méthodes.
Certaines méthodes peuvent être abstraites.

Une méthode abstraite comporte uniquement un en-tête.
Le corps de sa méthode n'est pas défini et est remplacé par un point-virgule.

```
Abstract class A
{
public void résolution()
{…
}
public abstract void affichage() ;
…..
}
```

Déclaration : A a ;
Instanciation impossible. A=new A(…) ; //pas possible

Pour créer des objets, il faut créer une sous-classe dans laquelle toutes les méthodes abstraites sont définies.

Cette classe qui n'est plus abstraite peut être instanciée.

Dérivation :

```
Class B extends A
{
public void affichage()
{…
}
…
}
```

Instanciation : B b= new B(..) ;

Egalement possible:

A a = new B(..);

8.9.2.: Propriétés:

Une classe qui possède au moins une méthode abstraite est abstraite : L'instanciation est impossible.
Elle devrait être déclarée avec le *mot-clé abstract* même si cette déclaration n'est pas obligatoire.
Une méthode abstraite doit obligatoirement être déclarée *publique.*

Dans l'en-tête d'une méthode abstraite, les arguments formels doivent être présents même s'ils n'ont aucune utilité.

Une classe dérivée d'une classe abstraite ne doit pas obligatoirement redéfinir toutes les méthodes abstraites de la classe de base.
Une classe dérivée qui comporte des méthodes abstraites de la classe de base non redéfinies, reste abstraite.

Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et/ou contenir des méthodes abstraites.
Cette propriété est utilisée implicitement puisque toutes les classes dérivent de Object.

8.9.3. : Objectifs des classes abstraites :

Les classes abstraites constituent une technique importante en POO.
Une classe abstraite permet de définir dans une classe de base, les méthodes communes aux différentes sous-classes tout en leur imposant de redéfinir les méthodes spécifiques. (sous-entendu dans les classes dérivées).
L'instanciation de cette super classe abstraite n'est pas possible, garantissant la sécurité d'avoir toutes les méthodes complètement définies.

8.10. : INTERFACES :

8.10.1. : Introduction :

L'interface possède un niveau d'abstraction supérieur à la classe abstraite.
Contrairement à la classe abstraite, l'interface est caractérisé par :

-des méthodes uniquement abstraites (aucune implémentation de méthodes).
-aucun champ, à l'exception des constantes.

8.10.2. : Propriétés :

Une classe peut implémenter plusieurs interfaces alors qu'une classe ne peut dériver que d'une seule classe abstraite.
Le concept d'interface se superpose à celui de dérivation et non s'y substitue.
Une interface peut se dériver.
Il est possible d'utiliser des variables d'interface.

Il est impossible de créer des objets de *type interface*.

8.10.3. : Définition d'une interface :

La définition d'une interface utilise le *mot-clé interface* à la place de *class.*
-les méthodes des interfaces sont abstraites(pas de définition de méthodes), et publiques.(redéfinitions ultérieures)

Ainsi, *les mot-clés public et abstract* peuvent être omis.

Exemple :

```
Public interface I
{
public abstract void résolution() ;
public abstract void affichage() ;
}
```

Une interface a les mêmes droits d'accès qu'une classe.
En général, une interface est publique.

8.10.4 : Implémentation d'une interface :

Une classe implémente une interface en utilisant le *mot-clé implements*

```
Class A implements I
{       //Redéfinition de résolution();
        //Redéfinition de affichage() ;
}
```

Il est impossible de différer les définitions de résolution et affichage, comme on pourrait le faire avec une classe abstraite.

Une même classe peut implémenter plusieurs interfaces.

```
Public interface I1
{
void résolution() ;
}

public interface I2
{
void affichage();
}

class A implements I1,I2
{
//Redéfinition de résolution();
//Redéfinition de affichage() ;
}
```

8.10.5. :Variable de type interface et polymorphisme :

Il est possible de définir des variables de type interface.

```
Public interface I
{
….
```

```
}
I i ;
```

Il est impossible d'instancier une interface, comme avec la classe abstraite.
Ainsi, il est impossible d'affecter à i une référence à un objet de type I.

Il est possible d'affecter à n'importe quelle référence à un objet d'une classe implémentant l'interface I.
Cette classe peut être quelconque, non nécessairement liée par l'héritage, du moment qu elle implémente l'interface I.

```
Public interface I
{…
}

class A implements I
{
…….
}
I i = new A (…);
```

8.10.6.: Interface et classe dérivée :

Le concept d'interface est indépendant du concept d'héritage.
Une classe dérivée peut implémenter une ou plusieurs interfaces.

8.10.7. : Interface et constante :

Une interface peut contenir des expressions constantes qui seront accessibles à toutes les classes implémentant l'interface.
Par définition, les expressions constantes sont *static et final*.
Ainsi, les *mot-clés static et final* peuvent être omis.

```
//Interface
interface EquationGeneral
{
static final int N=10;
…
}
```

```
//Interface
interface EquationGeneral
{
int N=100;
……
```

}

Evidemment, les constantes sont également accessibles en dehors de toute classe implémentant l'interface.

EquationGeneral N

Dans une interface, les méthodes doivent être implémentées dans les classes alors que les constantes sont utilisables par les classes.

/*NE SERA PAS UTILISE*/

8.10.8. : Dérivation d'une interface :

*Une interface peut être une généralisation d'une autre interface en utilisant le mot-clé extends : **dérivation d'interface ou héritage d'interface**.*

```
Interface I1
{
static final int N=1 ;
void résolution();
}

interface I2 extends I1
{
static final int M=100;
void affichage();
}
```

La dérivation des interfaces revient à concaténer les déclarations des interfaces . La définition de I2 est donc équivalente.

```
interface I2
{
static final int N=10 ;
static final int M=100;
void résolution();
void affichage();
}
```

8.10.9. : Conflits de nom (lors de la dérivation d'interface)

```
interface I1
{
```

```
void f(int n) ;
void g();
}

interface I2 extends I1
{
void f(double x);
void g() ;
}

class A implements I1, I2
{
…..
}
```

La classe A peut implémenter les 2 interfaces en redéfinissant les 2 méthodes void f(int) et void f(double) mais une seule méthode void g().

```
 interface I1
{
void f(int n) ;
void g();
}

interface I2 extends I1
{
void f(double x);
int g() ;
}

class A implements I1, I2
{
…..
}
```

IMPOSSIBLE : on ne peut pas avoir dans la même classe 2 méthodes g() : une définissant void et l'autre int. -> Question de polymorphisme.

8.11. CLASSES ENVELOPPES :

Les classes enveloppes permettent de convertir les types simples en type objets. Elles permettent d'avoir des méthodes et de compenser le fait que les types primitifs ne soient pas des classes.

<u>Les classes enveloppes sont les suivantes :</u>
Boolean, Byte, Character, Short, Integer, Long, Float et Double.

Construction d'une classe enveloppe :

ClasseEnveloppe(typeSimple)

Exemples :
Integer iObj= new Integer(10) ;
Double dObj= new Double(10.1)
iObj contient la référence à un objet de type Integer encapsulant l'entier 10.

Méthode permettant de retrouver la valeur dans le type primitif méthode réciproque)

TypeSimpleValue()

Exemples :
int i= iObj.intValue(); //i=10
double d= dObj.doubleValue(); //d=10.1

La méthode typeSimpleValue() permet de faire des conversions entre les différents types numériques (pas le booléen)

Integer iObj= new Integer(10)
Double d=iObj.doubleValue() ;
Int i=iObj.intValue(10)

8.12. : QUELQUES REGLES POUR LA CONCEPTION DE CLASSES :

L'héritage crée une relation de type « est » :
Si T' dérive de T, alors un objet de type T' peut être considéré comme un objet de type T.

public->private

La classe créé une relation de type « a » :
Si la classe T possède un champ de type U, alors un objet de type T possède un champ qui est un objet de type U.

/*Pas étudié*/

8.13.CLASSES ANONYMES :

Une classe anonyme est une classe sans nom temporaire.

Les classes anonymes sont principalement utilisées avec la gestion des évènements (écouteurs d'évènements).

(i) Classe anonyme dérivée d'une classe

```
A a ;
A=new A()
{
//Champs de la classe anonyme dérivée de A
//Méthodes de la classe anonyme dérivées de A.
…..
} ;
```

équivalent à :

```
class A1 extends A
{
………
}
A1 a =new A1();
```

Une classe anonyme ne peut pas introduire de nouvelles méthodes
Il est impossible de définir plusieurs références à classe anonyme.

(ii) Classe anonyme implémentant une interface.

IX . CHAINES DE CARACTERES.

9.1. : CHAINES DE CARACTERES (OBJET DE TYPE STRING) :

9.1.1. :Introduction :

La classe standard String permet de manipuler des chaînes de caractères (cdc).

Les cdc sont des objets de type String :
Elles ne sont pas des variables de type primitif.

Déclaration :

String ch

Ch est destinée à contenir une référence à un objet de type String.

Création automatique : " "

La notation Java désigne un objet de type String créé automatiquement par le compilateur.
String ch= ” java ” ;

La classe String possède 2 constructeurs:
-un constructeur vide créant une chaîne vide : String ch1 =new String()
-un constructeur avec un argument de type String : String ch2=new (” java ”) ;

String ch3=new String(ch2) : ch3 contient la référence à une chaîne copie de ch2.

Bien que les chaînes soient des objets, elles se manipulent comme des données de type primitif.
Il n’est donc pas nécessaire d’utiliser l’opérateur new.

9.1.2. : Valeur d’un objet de type String :

Un objet de type String n’est pas modifiable.(jamais)
Les références à des objets de type String peuvent être modifiées.
on peut modifier les références mais jamais le contenu.

On recopie la chaîne (avec StringBuffer) quand on a une grande chaîne ; on n’utilise jamais la classe String.

```
ChTemp=ch1 ;
Ch1=ch2 ;
Ch2=chTemp;
```

9.1.3.: Entrées/Sorties de chaînes :

La méthode println permet d’afficher des chaînes.

```
System.out.println(” java ”) ;
System.out.println(” ch ”) ;
```

Comme pour les autres types primitifs, il n’existe pas de méthode standard permettant de lire une chaîne au clavier.

9.1.4. : La méthode de longueur de chaîne : length() :

La méthode length() retourne le nombre de caractères de la chaîne.

```
Int i=ch.length() ;          // i donne la longuer de ch défini.
Int i=ch.length(”  ” ) ;     // i=0;
```

```
Int I=ch.length("java");      // i=1;
```

La longueur d'un tableau est donnée par le champ length.

9.1.5. : La méthode d 'accès aux caractères d'une chaîne :charAt :

Les caractères d'une chaîne sont indicés de 0 à (ch.length()-1) ,

```
String ch= "java " ;
C=ch.carAt(0);   //c='j'
C=ch.charAt(ch.length()-1) ;  c='a'
```

9.1.6.: L'opérateur de concaténation de chaînes + :

L'opérateur + permet la concaténation de 2 chaînes (objet de type String ou constante chaîne) en créant une nouvelle chaîne (un nouvel objet de type String).

L'opérateur + est associatif : ….+……..+……

9.1.7. :Conversion des opérandes de l'opérateur + :

L'opérateur + peut être utilisé avec un opérande de type primitif et un opérande de type chaîne.
La valeur de l'opérande de type primitif est automatiquement convertie en chaîne.

9.1.8. : L'opérateur de concaténation de chaîne +=.

L'opérateur de concaténation de chaîne += s'applique quand le 1er opérande est de type chaîne.

```
String ch= "langage";
Ch+= "java";                    // ch="langage java"
```

9.2. :METHODE DE RECHERCHE DANS UNE CHAINE : INDEXOF()

La méthode indexOf() de la classe String permet de trouver la 1ère occurrence d'un caractère ou d'une sous-chaîne donnée en argument, dans une chaîne.

Elle retourne :
-l'entier associé à l'indice du 1er caractère de la sous-chaîne si elle est trouvée.
-la valeur –1 sinon.

Public int indexOf(String str)

String ch="java";
Int I=ch.indexOf('j');   //I=0   on peut passer 'java' en type chaîne ou en type String
Int I=ch.indexOf("j");   //I=0

Public int indexOf(String str, int fromIndex)

Cette méthode permet de trouver la 1ère occurrence d'un caractère ou d'une sous-chaîne donnée en argument, à partir d'une position donnée en 2è argument d'une chaîne.

String ch="java";
Int I=ch.indexOf('v');   //I=2
Int I=ch.indexOf("v",3);   //I=-1

La méthode lastIndexOf() effectue les mêmes recherches que la méthode indexOf() mais en analysant la chaîne depuis sa fin.

Public int lastIndexOf (String str)
Public int lastIndexOf (String str, int fromIndex)

9.3.: METHODES DE COMPARAISON DE CHAINES :

9.3.1. : Les opérateurs == et != :

voir manuel

Comme les chaînes sont des objets, les opérateurs == et != comparent les références données comme opérandes…..

Remarque (QUESTION EXAM ??) :
== et != ne fonctionnent pas pour comparer 2 chaînes
-> utiliser alors la méthode equals(). Equals sur quelque chose qui n'est pas un objet de type String, il faut alors **redéfinir** la méthode equals de la super –classe objet comme on le fait pour la méthode toString().

9.3.2. : La méthode de comparaison de 2chaînes : equals() :

La méthode equals() de la classe String compare le contenu de 2 chaînes.

Public boolean equals(String anotherString)

```
Ch1.equals(ch2);
Ch1.equals("java");
```

La méthode equalsIgnoreCase() compare le contenu de 2 chaînes sans distinguer les majuscules des minuscules.

Public boolean equalsIgnoreCase(String anotherString)

Remarque:

```
Object o;
o.equals(…) ;  //méthode equals de Object.
               // comparaison de références.

String ch ;
Ch.equals(…) //méthode equals de String.
             //comparaison de valeurs.
```

9.9.3. : La méthode de comparaison de 2 chaînes compareTo() :

La méthode compareTo() de la classe String compare lexicographiquement le contenu de 2 chaînes.
Elle retourne :
- un entier négatif si la chaîne précède l'argument chaîne.
- Un entier nul si la chaîne et l'argument chaîne sont égaux.
- Un entier positif si la chaîne suit l'argument chaîne.

Public int compareTo(String anotherString)

9.4.: MODIFICATION DE CHAINES:

->elle est modifiée, mais création d'une nouvelle chaîne : il y aura 2 chaînes…

9.4.1. : La méthode de remplacement de caractères replace() :

La méthode replace de la classe String remplace toutes les occurrences d'un caractère donné par un autre en créant une nouvelle chaîne.

Public String replace(char oldChar, char newChar)

9.4.2.: La méthode d'extraction de sous-chaîne : subString() :

La méthode subString() de la classe String créé une nouvelle chaîne en extrayant de la chaîne courant :
-soit tous les caractères depuis une position donnée

public String subString (int beginIndex)

-soit tous les caractères compris entre 2 positions données, la 1ère position étant incluse et la 2ème position étant exclue.

Public String subString(int beginIndex, int endIndex)

9.4.3.:La méthode de passage en majuscule ou minuscule : toLowerCase() et toUpperCase() :

La méthode toLowerCase de la classe String créé une nouvelle chaîne en remplaçant toutes les majuscules par leur équivalent en minuscules.

Public String toLowerCase()

La méthode toUpperCase() de la classe String créé une nouvelle chaîne en remplaçant toutes les minuscules par leur équivalent en majuscules.

Public String toUpperCase()

9.5.TABLEAU DE CHAINES:

Il est possible de créer des tableaux d'objets, donc en particulier des tableaux de chaînes.

```
Public static void main(String args [ ])
{
        String tabCh [ ]={"Fortran","Pl1",....);
                for(int i=0;i<tabCh.length;i++)
                        System.out.println(tabCh[i]);
}
```

Exécution:
Fortran
Pl1
....

```
Public static void main(String args [ ])
{
        String tabCh [ ]={"Fortran","Pl1",....);
                for(int i=0;i<tabCh.length;i++)
                {
                 for(int j=0;j<tabCh[i].length(); j++)
                        System.out.println(tabCh[i].charAt(j));
                                System.out.println();
                }
}
```

Même exécution…

Remarque : (QUESTION EXAM ??)
Si on a un tableau de chaînes (de type char) ou une chaîne.
->Quelle version choisir. Si oui pourquoi ?

on ne prend en aucun cas, PLUS un tableau de caractères (car limité)
-> remplacé par le type chaîne (grâce à toutes les méthodes déjà implémentées et pas de problème de limite).

9.6. : CONVERSIONS ENTRE CHAINES ET TYPES PRIMITIFS :

9.6.1 : Conversions d’un type primitif ou objet, en une chaîne :

L’opérateur + convertit n’importe quel type primitif ou objet, en une chaîne .
Une telle conversion peut être effectuée directement par la méthode statique ValueOf de la classe String :

Public static String valueOf(Type_primitif p)
Public static String valueOf(Object obj)

Exemples:
String ch=String.valueOf(p);   c’est une méthode statique.Donc on appelle String.
String ch=String.valueOf(obj);

L’affectation : ch=String.valueOf(n) est équivalente à: ch=” ”+n ;

(utilisation artificielle d’une chaîne vide pour la conversion de l’opérateur +)

L’expression : String.valueOf(obj) est équivalente à : Obj.toString()

Comme les types primitifs possèdent des classes enveloppes, l’expression

String.valueOf(n)

Est équivalente à

Integer(n).toString(n)

Remarques :
La conversion d’un type primitif ou objet, en une chaîne, aboutit toujours .
La conversion d’un réel flottant en chaîne peut conduire à garder ou à perdre le symbole puissance E.

9.7. :CONVERSIONS ENTRE CHAINES ET TABLEAUX DE CARACTERES :

9.7.1. : Conversion d’un tableau de caractères en chaîne :

On utilise le constructeur de la classe String

Public String(char[ ] value)

Exemple:

Char tab[ ] ={‘j’,’a’,’V’,’a’};

String ch= new String(tab);

Autre constructeur de la classe String qui ne considère qu'un certain nombre de caractères à partir d'une position donnée.

Public String (char[ ] value, int offset, int count)

9.6.2.: Conversion d'une chaîne en type primitif:

La conversion d'une chaîne de type primitif fait appel à une méthode de la classe enveloppe associée au type primitif.
La conversion n'aboutit pas systématiquement (exception de type NumberFormatException).

Pour la classe enveloppe Integer :

Public static int parseInt(String s) throws NumberFormatException

Exemple:
int n = Integer.parseInt(s);

Pour la classe enveloppe Long:

Public static long parseLong(String s) throws NumberFormatException

Pour la classe enveloppe Double:

Public static double parseDouble(String s) throws NumberFormatException

Remarques:

-Le signe + n'est pas accepté par les méthodes de conversion en entier.
-le signe + est accepté par les méthodes de conversion en réel.

9.7.2. : Conversion d'une chaîne en tableau de caractères :

On utilise une méthode de la classe String :

Public char[ ] toCharArray()

Exemple:
String ch;

Char tab[ ] = ch.toCharArray();

Autre méthode de la classe String:

Public void getChars(int scrbegin, int scrEnd, char[ ] dst, int dstBegin)

9.8.:ARGUMENTS DE LA LIGNE DE COMMANDE:

L'en-tête de la méthode main se présente de la façon suivante :

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

La méthode main reçoit un argument de type tableau de chaînes, destiné à contenir les éventuels arguments fournis au programme lors de son lancement à partir d'une ligne de commande.

Technique utilisée avec les environnements de développement intégré.

9.9. :LA CLASSE StringBuffer :

->utilisée quand on a une longue chaîne.

Les objets de type String ne sont pas modifiables.
Leur manipulation conduit à la création de nouvelles chaînes.
Dans les programmes manipulant intensivement les chaînes, il existe une perte de temps importante.
La classe StringBuffer permet de manipuler les chaînes en modifiant les objets

3 constructeurs :

```
public StringBuffer()
public StringBuffer(int length)
public StringBuffer(String s)
```

Création d'un objet de type StringBuffer à partir d'un objet de type String:
String ch ;
StringBuffer chBuf=new StringBuffer(ch) ;
|->pour le rendre modifiable

Principales méthodes de la classe StringBuffer :

-append(…) : ajout d'une chaîne en fin.
-charAt(…) : accès à un caractère de rang donné.
-delete(…) : suppression d'une sous-chaîne.
-deleteCharAt(…) : suppression d'un caractère.
-getChars(…) : copie d'une sous-chaîne dans un tableau de caractères..
-insert(…) : insertion d'une chaîne à une position donnée.
-length(…) : longueur de la chaîne de type StringBuffer.
-replace(…) : remplacement d'une sous-chaîne par une autre.
-reverse(…) : renverse la chaîne.
-setCharAt(…) : modification d'un caractère de rang donné.
-setLength(…) : fixe la longueur de la chaîne de type StringBuffer.
-subString(…) : retourne une sous-chaîne.
-toString(…) : conversion en objet de type String.

XI. LES FLUX

11.1 : INTRODUCTION :

Java utilise la notion de flux (canal)

Un flux de sortie est un canal quelconque capable de recevoir de l'information sous forme d'une suite d'octets : périphérique d'affichage, fichier, connections à un site distant, un emplacement en mémoire centrale.

Un flux d'entrée est un canal quelconque capable de délivrer de l'information sous forme d'une suite d'octets : périphérique de saisie, fichier, connexion à un site distant, un emplacement en mémoire centrale.

Un flux binaire transmet l'information sans modification de la mémoire au flux et réciproquement.

Un flux texte transmet l'information avec modification appelée formatage pour que le flux reçoive ou transmette une suite de caractères, par exemple la méthode println réalise un tel formatage.

11.2. : LES FLUX TEXTE.

11.2.1. : Généralités :

Un fichier texte peut être :

-créé ou lu avec un éditeur de texte ou un traitement de texte en mode texte.

-listé par une commande de l'environnement, type sous DOS, *more ou pr* sous UNIX.

Chaque caractère dans un fichier texte est codé sur un seul octet et suivant un code dépendant de l'environnement.

Le caractère de fin de ligne est représenté par 2 caractères : CR(code héxadécimal 13) et LF (code héxadécimal 10) dans l'environnement PC, et un caractère LF (code héxadécimal 10) dans l'environnement UNIX.

Les flux texte doivent subir des transformations :

-pour un flux teste de sortie :

conversion de 2 octets représentant un caractère Unicode en un octet correspondant au code local de ce caractère dans l'environnement.

-pour un flux texte en entrée :

conversion d'un octet représentant un caractère dans le code local de l'environnement en 2 octets correspondant au caractère Unicode.

-transformation du caractère de fin de ligne selon leur représentation locale.

11.2.2. : Ecriture d'un fichier texte :

La classe abstraite Writer est la classe de base à toutes les classes relatives à un flux texte de sortie.
La classe FileWriter, dérivée de OutputStreamWriter qui dérive de Writer, est la classe pour la création d'un flux texte de sortie associé à un fichier.

Ouverture d'un fichier texte en écriture :

Un constructeur :

```
Public FileWriter(String fileName) throws IOException ;
FileWriter fw= new FileWriter("fichier.txt");
```

L'objet fw est associé à un fichier de nom fichier.txt. Si le fichier n'existe pas, alors il est créé. S'il existe, son ancien contenu est détruit.

Ouverture d'un fichier texte en écriture avec formatage avec la classe PrintWriter qui possède des méthodes println et print :

Un constructeur :

}

```
Public PrintWriter(FileWrtiter fw)
PrintWriter f= new PrintWriter(fw)
OU
PrintWriter f = new PrintWriter(new FileWriter(“fichier.txrt”));
```

Fermeture d’un fichier texte en écriture:

Avec import.java.io.* ;

Méthode : public void close()

```
//Méthode d’affichage
public void affichage()
{
DecimalFormat deuxDecimal = new DecimalFormat («0.00 ») ;
System.out.println(“ Equation= “
                    +deuxDecimal.format(coeffX2)+ « x2 +»
                    +deuxDecimal.format(coeffX1)+ « x + »
                    +deuxDecimal.format(coeffX0) ;
System.out.println(«        Racine1= » +deuxDecimal.format(racine1)) ;
System.out.println(«        Racine2= » +deuxDecimal.format(racine2)) ;
}
```

```
//Méthode d’affichage dans un fichier
public void affichageFichier()
{
PrintWriter f=null ;

Try
{
f= new PrintWriter (new FileWriter(“Equation.txt”);

DecimalFormat deuxDecimal= new DecimalFormat(“0.00”);

f.println(“Equation=
             +deuxDecimal.format(coeffX2)+ « x2 +»
             +deuxDecimal.format(coeffX1)+ « x + »
             +deuxDecimal.format(coeffX0)) ;

f.println(« Racine1= »+deuxDecimal.format(racine1)) ;
f.println(« Racine2= »+deuxDecimal.format(racine2)) ;
}

catch (IoException e)
{
System.out.println(“Erreur:” +e);
```

```
}
f.close();
}
}
```

11.2.3.: Lecture d’un fichier texte sans accès à l’information :

Il n’existe pas de classe de lecture symétrique à la classe PrintWriter.
Il faut utiliser la classe FileReader.

Ouverture d’un fichier texte en lecture :

Un constructeur :

```
Public FileReader(String fileName) throws FileNotFoundException ;
FileReader fr = new FileReader(“fichier.txt”);
```

FileReader ne peut accéder qu’à des caractères et nécessite la gestion de la fin de ligne. Il faut l’associer à la classe BufferReader qui possède une méthode readLine.

Un constructeur :

```
Public BufferedReader(FileReade fr)
BufferedReader f =new BufferedReader(fr)
OU
BufferedReader f = new BufferedReader (new FileReader(“fichier.txt”))
```

La méthode readLine de la classe BufferedReader fournit une référence à une chaîne correspondant à une ligne de fichier.

Fermeture d’un fichier texte en lecture :

Méthode : public void close( ) throws IOException.
La méthode close doit être incluse dans un bloc try-catch.

```
Package coursjava ;
Import java.io.* ;

Public class LectureFichier
{
//Méthode principale
public static void main(String[ ] args)
{
```

```
String ligne;
BufferedReader fr;

Try
{
fr= new BufferedReader( new FileReader("Equation.txt"));

while((ligne=f.redline()) !=null)
{
System.out.println(ligne);
}
f.close();
}
catch(IOException e)
{
System.out.println("ERREUR:"+e);
}
}
}
```

11.2.4.: Lecture d'un fichier texte avec accès à l'information:

La classe StringTokenizer permet de découper une chaîne en différents tokens (sous-chaîne) en se basant sur des caractères séparateurs choisis.
Il est possible d'appliquer à ces différents tokens des conversions en type primitif.

Constructeur :

Public String Tokenizer(String str,String delim)

Méthodes:
(dans les boucles)
-countTokens : compte le nombre de tokens.
-nextToken : donne le token suivant s'il existe en retournant le type chaîne.
-nextElement : donne le token suivant s'il existe en retournant le type objet.

```
Package coursjava ;

Import java.io.* ;
Import java.util.* ;

Public class LectureFichier
{
public static void main(String[ ] args)
{
int nbTok=0;
```

```
double x;
String ligne=" ";
BufferedReader f;

Try
{
f= new BufferedReader (new FileReader("Equation.txt"));

while((ligne=f.readline())!=null)
{
System.out.println("Fichier:" +ligne);
StringTokenizer  tok = new StringTokenizer(ligne, « »);
NbTok = tok.countTokens();

For (int  i=0;i<nbTok;i++)
{
x=Double.parseDouble(tok.nextToken());
System.out.println("Token:"+x+" ");
}
Syqtem.out;println();
}
f.close();
}
catch (IOException e)
{
System.out.println("ERREUR: "+e);
}
}
}
```

Exécution:

```
Fichier:   1.1
Token:1.1
Fichier 2.22   3.333
Token:2.22   Token :3.333
Fichier 4.4444      5.55555     6.666666
Token:4.4444      Token :5.55555    Token :6.666666
```

11.3.: DES FLUX BINAIRES

11.4.: GESTION DES FICHIERS AVEC LA CLASSE FILE

11.5. :DESCRIPTION DES CLASSES FLUX

XII. LA CLASSE java.lang.Math :

La classe ne comporte que des champs statiques et des méthodes statiques.
Il ne faut pas confondre la classe java.lang.Math avec le paquet java.math qui permet de travailler avec des nombres importants et de grande précision.

12.1. : CHAMPS STATIQUES DE LA CLASSE java.lang.math. :

-E
-PI

12.2. : METHODES STATIQUES DE LA CLASSE java.lang.math. :

Sauf exception, le type de retour des méthodes est identique à celui des arguments.

-abs(double a)
-abs(float a)
-abs(float a)
-abs(int a)
-abs(long a)
-acos(double a)
-asin(double a)
-atan(double a)
-floor(double a)
-exp(double a)
-max(double a, double b)
-max(float a, float b)
-max(int a, int b)
-max(int a, int b)
-max(long a, long b)
-min(double a, double b)
-min(float a, float b)
-min(int a, int b)
-min(long a, long b)
-pow(double a, double b)
-random( )

12.3.: CLASSE RANDOM DE java.util:

La classe Random de java.util permet des possibilités étendues de génération de nombres aléatoires.

Constructeur :

-Random( ):génération de nombres aléatoires en fonction de l'horloge interne (séquences aléatoires différentes)
-Random (long seed) : génération de nombres aléatoires d'une valeur d'amorce (séquences aléatoires identiques).

Méthodes :
-int next (int bits) : generates the next pseudorandom number.
-boolean nextBoolean(): returns the next pseudorandom, uniformly distributed boolean value from this random bytes and places them into a user supplied byte array.
-double nextDouble();
-double nextGaussian():returns the next pseudorandom, Gaussian("normally") distributed double value with mean 0.0 and standard deviation 1.0 from this random number generator's sequence.
-int nextInt(int n)

-void setSeed(long seed);

XIII. STRUCTURES DES DONNEES DYNAMIQUES :

13.1. : Les structures de données :

Les structures de données dynamiques différentes des structures de tailles fixe, ont une taille qui se modifie au cours de l'exécution.

Ces structures dynamiques sont particulièrement adaptées aux cas où le nombre d'éléments de données à représenter dans la structure de données est inconnu.

Il existe différents types de structures de données :

-Les listes chaînées : collection d'éléments de données qui sont chaînées, permettant l'insertion et les suppressions en n'importe quelle position.
La classe LinkedList du package java.util permet de manipuler les listes chaînées.

-les piles :

-les files : (queues, FIFO :First In First Out) : cas particulier des listes chaînées avec des insertions à une extrémité appelée fin de la file et des suppressions à l'autre extrémité appelée tête de la file.

-les arbres, en particulier binaires : structure de données non linéaire.

Le principe de programmation de ces structures de données repose sur le concept de classe autoréférentielle.

13.2. : Classe autoréférentielle :

Une classe autoréférentielle contient un membre qui consiste en une référence à un objet d'une classe de même type de classe.

```
//Classe autoréférentielle
class Nœud
{
private int donnee ;
private Nœud suivant ;
.
.
.
```

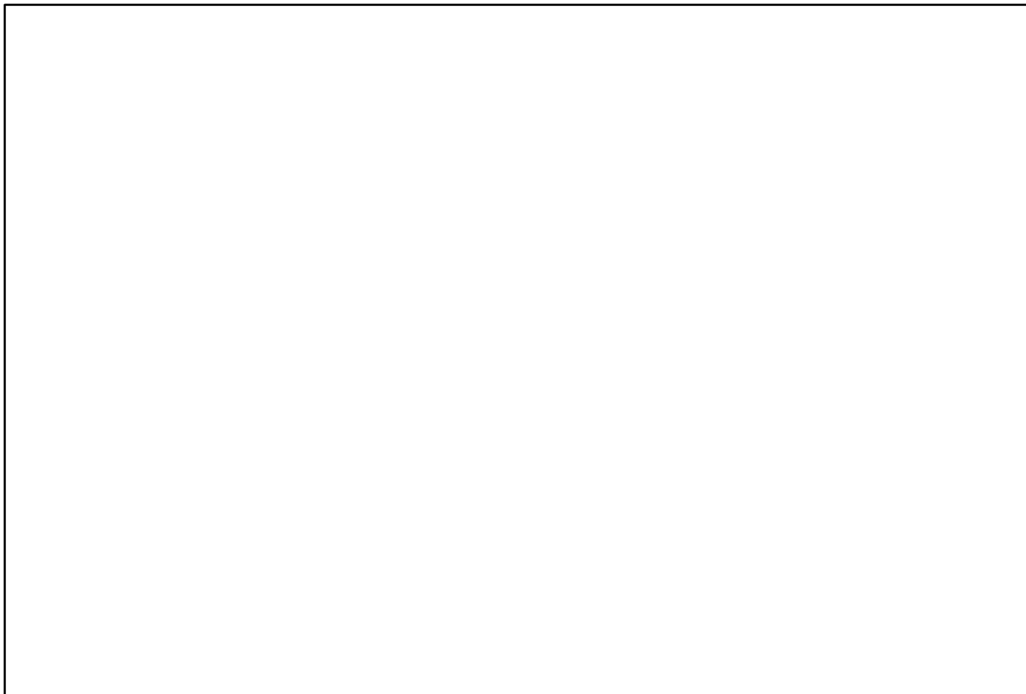
Cette classe définit un Nœud avec 2 variables d'instance private : l'entier *donnée* et une référence à un Nœud nommé *suivant* .

Classe autoréférentielle à l'automate à pétales :

```
Class Nœud
{
final int DIM=4 ;
Object sommet;                    //étiquette des noeuds
Object arc[ ] = new Object[DIM];   //tableau d'arc(il y en a 4 : A C G T)
Nœud succ[ ] = new Nœud [DIM] ; //tableau de successeurs
Public  Nœud (Object o)
{
sommet=o ;

for(int i=0,i<DIM ;i++)
{arc[I]=null;
succ[I]=null;
}
}

public String toString()
{
return sommet.toString();
}
}
```



13.3.: Liste Chaînée:

La liste chaînée est une collection linéaire d'objets d'une classe autoréférentielle appelés nœuds et reliés par des liens.

L'accès à une liste chaînée s'effectue par une référence au 1er nœud de la liste. Chaque nœud successif est accessible par le membre de référence de lien stocké dans le nœud précédent.

La référence de lien dans le dernier nœud est mise à null et marque la fin de la liste.

Un nœud contient des données de tout type, y compris des objets d'autres classes.

Programme qui présente des sous-chaînes d'une chaîne en éléments d'une liste chaînée.

```
Package coursjava ;
//- -------
class NoeudListe
{
Object element ;
NoeudListe suivant ;

Public NoeudListe (Object élément, NoeudListe suivant)
{
this.élément=élément ;
this.suivant=suivant ;
}


public NoeudListe (Object élément)
{
this (élément, null) ;
}

public NoeudListe()
{
this(null,null) ;
}

//Méthode qui retourne une référence à l'élément de ce nœud.
public Object getElement( )
{
return élément;
}

//- ---------
class Liste
{
private NoeudListe premierNoeud;
private NoeudListe dernierNoeud;
private String nomListe;

public Liste (String nomListe)
```

```
{
this.nomListe=nomListe;
}

public Liste()
{this("Liste");
}

public boolean listeEstVide
{
return (premierNoeud = = null);
}

public void ajouterEnQueue (Object element)
{
if (listeEstVide())
{
premierNoeud =dernierNoeud= new NoeudListe (element) ;
}
else
{
dernierNoeud = dernierNoeud.suivant= new NoeudListe (element) ;
}
}
```

```
//Méthode de suppression d'un élément quelconque de la liste.
Public void suppressionElement(Object element)
{
	NoeudListe courant=premierNoeud ;
	NoeudListe precedentCourant=premierNoeud ;


	while((courant.suivant !=null)&&( !(courant.element.equals(element))) ;
	{
	precedentCourant=courant ;
	courant=courant.suivant ;
	}


//Suppression de l'élément.
if((courant.element.equals(element))&&(courant !=premierNoeud))
{
	precedentCourant.suivant=courant.suivant;
}

else
{
```

```
//suppression du 1er elément
if (courant==premierNoeud)
{
	premierNoeud=courant.suivant;
}

else
{
	System.out.println(element+ »n'existe pas dans la liste »+nomListe) ;
}

//Méthode pour insérer en tête de liste
public void insererEnTete(Object element)
{

if (listeEstVide())
	premierNoeud=dernierNoeud=new NoeudListe(element) ;

else
	premierNoeud=new NoeudListe(element,premierNoeud) ;
}
```

```
//Méthode pour insérer en queue de liste :
public void insererEnQueue(Object element)
{

if (listeEstVide())
	premierNoeud=dernierNoeud=new NoeudListe (element) ;

else
	dernierNoeud=dernierNoeud.suivant= new NoeudListe(element) ;
}
```

13.4. : PILE :

2 méthodes qui interviennent dans une pile :

- -push( )
- -pop( )

13.5. : FILE :

2 méthodes qui interviennent dans une pile :

- -enQueue( ) : ajoute un nouveau nœud en fin de file
- -deQueue( ) : supprime un nœud en tête de file.

13.6. :ARBRE BINAIRE :

Un arbre binaire est un arbre dont tous les nœuds comportent au plus deux liens. Le 1er nœud de l'arbre est appelée racine.

-Chaque lien du nœud racine fait référence à un enfant.
-L'enfant gauche est le tout 1er nœud du sous-arbre gauche et l'enfant droit est le tout 1er nœud su sous-arbre droit.
-Les enfants d'un nœud dont appelées frères.
-Un nœud sans enfant est appelé feuille.

L'arbre de recherche binaire est un arbre binaire particulier tel que :
-les valeurs de tout sous-arbre gauche sont < à la valeur de leur nœud parent, et que
-les valeurs de tout sous-arbre droit sont > à la valeur de leur nœud parent.

On suppose également qu'il n'existe pas de valeurs identiques.

Remarque : L'arbre de recherche binaire correspond à une série de données qui peut varier selon l'ordre d'insertion des valeurs.

3 TYPES DE PARCOURS :

Parcours en ordre.
-parcourir le sous-arbre gauche avec un appel à parcours en ordre.
-traiter le nœud.
-parcourir le sous-arbre droit avec un appel à parcours en ordre.

Le parcours en ordre d'un arbre de recherche binaire affiche les valeurs des nœuds dans l'ordre croissant : tri par arbre binaire.

Parcours en pré-ordre.
-traiter le nœud.
-parcourir le sous-arbre gauche avec un appel à parcours en pré-ordre.
-parcourir le sous-arbre droit avec un appel à parcours en pré-ordre.

Parcours en post-ordre.
-parcourir le sous-arbre gauche avec un appel à parcours en post-ordre.
-parcourir le sous-arbre droit avec un appel à parcours en post-ordre.
-traiter le nœud.

```
Class NoeudArbre
{
```

```
int element ;
NoeudArbre  gauche ;
NœudArbre droit ;


public NoeudArbre(int element)
{
	this.element=element ;
	gauche=droit=null ;
}


public void insérer(int element)
{
	if (element < this .element)
	{
		if (gauche= =null)
		{
			gauche= new NoeudArbre(element);
		}

		else
		{
			gauche.inserer(element) ;
		}
	}

	else
	{
		if (element>this.element)
		{
			if(droit= =null)
		{
			droit= new NoeudArbre(element);
		}

		else
		{
			droit.inserer(element) ;
		}
	}
}
```

```
class Arbre
{
private NoeudArbre racine;

public Arbre()
{racine=null;
}
```

```
public void insererNoeud(int element)
{
        if (racine= = null)
        {
        racine= new NoeudArbre(element);
        }
        else
        {
        racine.insere(element);
        }
}

public void enOrdre(NoeudArbre nœud)
{
        if (nœud= =null)
        {
        return;
        }

        enOrdre(noeud.gauche);
        System.out ;println(nœud.element+ » «) ;
        EnOrdre(nœud.droit) :
}
```

Il existe encore d'autres méthodes mais elles fonctionnent selon le même principe.

13.7. : CLASSE VECTOR :

La classe Vector de java.util dérive de l'Interface List.
Elle permet de créer des objets du type tableau qui croissent et décroissent dynamiquement en fonction des besoins, contrairement aux tableaux statiques dont la taille est fixée à la compilation.

Les éléments dans Vector sont des objets. Ainsi, avec des types primitifs, il faut utiliser les classes enveloppes.

Constructeurs :

- -Vector( )
- -…

Méthodes :

- -void add(int index, Object elem)
- -…..

Programme avec la classe Vector:

```
import.java.util.* ;

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

```
{
Vector vect = new Vector( );

        for (int i=0; i<10; i++)
        {
                vect.add(new Integer(i));
        }

System.out.println(“Suite d’entiers dans le vecteur:”);

        for (int i=0; i<vect.size( ); i++)
        {
                System.out.println(vect.elementAt(i)+” “);
        }

System.out.println( ) ;
System.out.println(“firstElement( ):”+vect.firstElement( ));
System.out.println(“lastElement( ):”+vect.lastElement( ));

vect.remove(o);

System.out.println(”Suite d’entiers dans le vecteur: “);



        for (int i=0; i<vect.size( ); i++)
        {
                System.out.println(vect.elementAt(i)+” “);
        }

System.out.println( );

Vect.removeElement(new Integer(5));

System.out.println(”Suite d’entiers dans le vecteur: “);

        for (int i=0; i<vect.size( ); i++)
        {
                System.out.println(vect.elementAt(i)+” “);
        }
}
```

EXECUTION:

Suite d’entiers dans le vecteur: 0 1 2 3 4 5 6 7 8 9
FirstElement( ) : 0
LastElement( ) :9
….

13.8. : CLASSE STACK :

La classe Stack de java.util qui dérive de Vector permet de manipuler la pile.
Elle étend la classe Vector.
Les éléments de Stack sont des objets. Avec des types primitifs, il faut alors utiliser les classes enveloppes.

<u>Constructeurs :</u>

-Stack( )
-….

<u>Méthodes :</u>

-boolean empty( )
-Object peek( )

Programme utilisant la classe Stack :

```
Import.java.util.*

Public class TestStack
{
	public static void main (String [ ] args)
	{
	Stack pile = new Stack( );

		for (int i=0; i<10; i++)
		{
			pile.push(new Integer(i));
		}

	System.out.println("Suite d'entiers dans la pile:");

		for (int i=0; i<pile.size( ); i++)
		{
			System.out.println(pile.elementAt(i)+" ");
		}

	System.out.println( ) ;
	System.out.println("peek( ):"+pile.peek( ));
```

```
        Object elementEnleve= new object( ));

        ElementEnleve=pile.pop( );

.
.
.
.
```

13.10.2. : Classe Arrays:

La classe Arrays fournit des méthodes de manipulation de tableaux :

-tri(sort) ,
-recherche dans un tableau trié (binarySearch),
-comparaison de tableaux (equals),
- placement d'éléments dans un tableau (fill)
-asList( ) qui permet de voir un tableau comme une liste (vue de liste)

```
import.java.util.*

public class TestArrays
{
public static void main (String [ ] args)
{
Object t[ ] = new Object [10];
Random r= new Random (99);

Arrays.fill(t, new integer(0));
…
for (int I=0; I< t.length; I++)
{
        t[I] = new Integer (r. nextInt(100));
}
.
.
.
Arrays.sort(t);

System.out.print(« Suite d'entiers dans le tableau : ») ;

For(int i =0 ; i< t.length ; i+=)
{
        System.out.print(t[I]+" ");
}
….

//Méthode binarySearch utilisable uniquement après la méthode sort.
```

```
Int rechercheInt = Arrays.binarySearch (t, new Integer(74)) ;

System.out.print(Position : »+rechercheInt) ;

//Création d'une liste de taille fixe :
List laListe = Arrays.asList(t) ;
System.out.println( ) ;

        For(int i=0 ;i< laListe.size( ) ; i++)
        {
                System.out.println(laListe.get(i)+ '' '' ) ;
                System.out.println( ) ;
        }
}
```

EXECUTION :
Suite d'entiers dans le tableau : 0 0 0 0 0 0 0 0
……   :87 58 29 11 0 …..

13.10.3. : Interface Collection : (non traité)

13.10.4. : Classe Collections :

Elle fournit des méthodes de manipulation de collections : algorithme de recherche, tri, ……

Méthodes :

-int binarySearch(List liste, Object key)
-void copy (List dest, ______)

……

13.10.5.: Interface List:

Une List est une Collection ordonnée pouvant contenir des éléments en double. L'Interface List est implémentée par les classes ArrayList,……

13.10.6. : Classe ArrayList :

Elle dérive de List . C'est une liste de tableau. Les méthodes de cette classe sont similaires aux méthodes de Vector mais sans le concept de synchronisation.

Constructeurs :

-ArrayList( ) ;

-……..

Programme utilisant la classe ArrayList :

```
Import.java.util.* ;

Class EnsembleConstruction
{
public ArrayList ens ;

public EnsembleConstruction (String numEns)
{
ens = new ArrayList() ;
Syqtem.out.println("Ensemble"+numEns+"créé");
}

public boolean ensembleContientElement (string numEns, Object elem)
{
boolean elementExiste ;

elementExiste= ens.contains(element) ;

System.out.println(…) ;
}
```

13.10.7. : Classe LinkedList :

Elle dérive de List et permet de manipuler des listes chaînées.

Constructeurs :

-LinkedList( ) ;
-LinketList(Collection c)

…..

13.10.8.: Classe Set:

C'est une Collection ne comportant que des éléments uniques.

2 implémentations :

-HashSet : stocke ses éléments dans une table
-treeSet

13.10.9. : Interface Map : (non traité).