PROGRAMMATION ORIENTEE OBJET
JAVA
Programmes du cours

Christian MICHEL

Université Louis Pasteur Strasbourg
Département Informatique

michel@dpt-info.u-strasbg.fr

PLAN

1 INTRODUCTION

1.1 HISTORIQUE

1.2 JAVA ET LA PROGRAMMATION ORIENTEE OBJET (POO)

1.2.1 Objet

1.2.2 Encapsulation des données

1.2.3 Classe

1.2.4 Héritage

1.2.5 Polymorphisme

1.2.6 Langage de POO presque pur

1.3 JAVA ET LA PROGRAMMATION EVENEMENTIELLE

1.4 JAVA ET LA PORTABILITE

1.5 AUTRES PARTICULARITES DE JAVA

1.6 PHASES DE DEVELOPPEMENT D'UN PROGRAMME JAVA

Phase 1: Edition
- Editeur
    + vi et emacs sous UNIX
    + Bloc-notes sous WINDOWS
    + Environnements de Développement Intégrés (EDI): JBuilder de Borland, NetBeans, Visual Cafe de Symantec, Visual J++ de Microsoft
- 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 bytes 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 de fichier.
Java est sensible à la casse.

Phase 3: Chargement
- Le chargeur de classe prend le ou les fichiers .class et les transfère en mémoire centrale
- Le fichier .class peut être chargé à partir d'un disque dur de sa propre machine ou à travers un réseau
- 2 types de fichier .class peuvent être chargés: les applications (programmes exécutés sur sa propre machine) et les applets (programmes stockés sur une machine distante et chargés dans le navigateur Web).

- **Une application** peut être chargée et 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 et exécutée par

+ le chargeur de classe lancé par le navigateur Web lorsqu'une applet est référencée dans un document HTML. Puis, l'interpréteur Java du navigateur pour exécuter l'applet

+ la commande appletviewer du J2SDK qui requiert également un document HTML pour exécuter l'applet. Exemple: appletviewer Programme.html avec un fichier Programme.html faisant référence à l'applet Programme.

Phase 4: Vérification

Les byte codes dans une applet sont vérifiés par le vérificateur de byte codes avant leur exécution par l'interpréteur Java intégré au navigateur ou à l'appletviewer. Ce vérificateur vérifie que les byte codes sont conformes aux restrictions de sécurité de Java concernant les fichiers et la machine.

Phase 5: Exécution

L'ordinateur interprète le programme byte code par byte code.

Les interpréteurs présentent des avantages sur les compilateurs dans le monde Java. En effet, un programme interprété peut commencer immédiatement son exécution dès qu'il a été téléchargé sur la machine cliente, alors qu'un programme source devant subir une compilation supplémentaire entraînerait un délai de compilation avant de pouvoir démarrer son exécution.

Cependant, dans des applets à forte charge de calcul, l'applet doit être compilé pour augmenter la rapidité d'exécution.

Documentation technique Java de Sun est disponible à l'adresse

http://java.sun.com

2 GENERALITES

2.1 PROGRAMME ECRITURE 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");
  }
}
```

Exécution:
Un programme Java

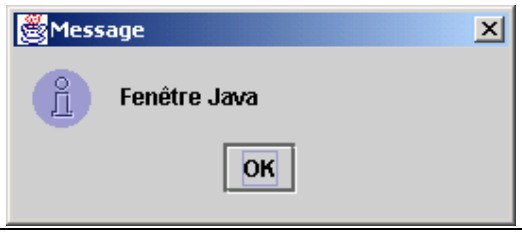
2.2 PROGRAMME ECRITURE FENETRE

Problème: Ecriture d'un texte dans une fenêtre graphique
Fichier: EcritureFenêtre.java

```
import javax.swing.*;

public class EcritureFenêtre
{
  public static void main(String[] args)
  {
    JOptionPane.showMessageDialog(null,"Fenêtre Java");
  }
}
```

Exécution

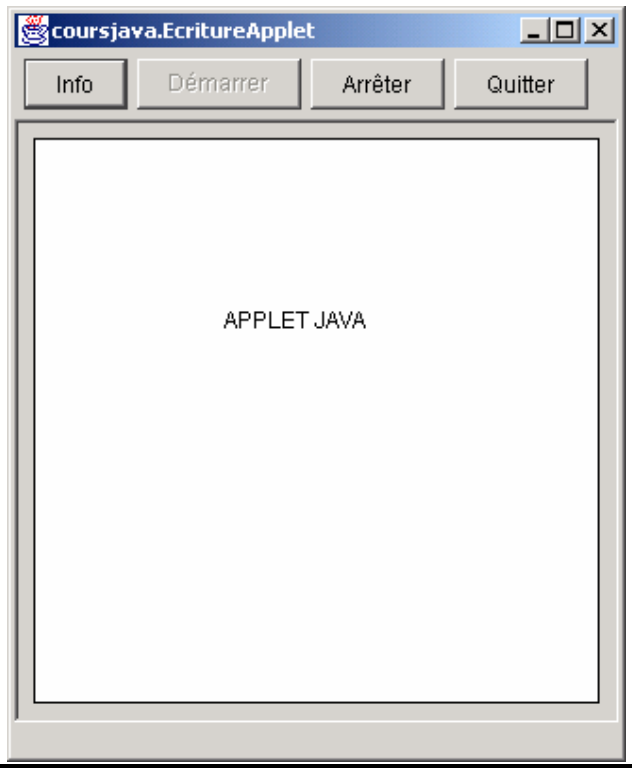


Problème: Ecriture d'un texte dans une applet
Fichier: EcritureApplet.java

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

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

Exécution


coursjava.EcritureApplet
Info
Démarrer
Arrêter
Quitter
APPLET JAVA

2.3 PROGRAMME LECTURE

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

```
import java.io.*;

// Méthodes 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 double lireDouble()
{
  double x = 0;

  try
  {
    String ligne_lue = lireString();
    x = Double.parseDouble(ligne_lue);
  }
  catch (NumberFormatException err)
  {
    System.out.println("Erreur de donnée");
    System.exit(0) ;
  }
  return x;
}

// 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 donnée");
    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);
    System.out.print("Donner un entier: ");
    int n;
    n = Lecture.lireInt();
    System.out.println("Résultat " + n);
  }
}
```

Exécution

Donner un double: 10.01
Résultat 10.01
Donner un entier: 10
Résultat 10

2.4 REGLES GENERALES D'ECRITURE

2.4.1 Identificateurs

Par convention
Les identificateurs de classes 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, comporte une majuscule à chaque début de mot sauf pour le 1er mot qui dépend du type d'identificateur.
Exemple: public class ClasseNouvelle

2.4.2 Mots clés

2.4.3 Séparateurs

2.4.4 Format libre

Par convention: une instruction par ligne.

2.4.5 Commentaires

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

2.4.6 Blocs

2.4.7 Code Unicode

Java utilise le système Unicode pour coder les caractères. Chaque caractère Unicode est codé sur 2 octets conduisant à 2^{16}=65536 possibilités qui permettent de représenter la plupart des alphabets (latin, grec, cyrillique, arménien, hébreu, arabe, etc.) et des symboles mathématiques et techniques.

3 TYPES PRIMITIFS

3.1 NOTION DE TYPE

3.2 TYPE BOOLEEN

Déclaration d'une variable booléenne
boolean test

Une variable booléenne prend 2 valeurs: false et true.

Affectation d'une variable booléenne
test = (n < m)

3.3 TYPE ENTIER

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

Type	Taille (octet)	Valeur minimale	Valeur maximale
int	4	-2 147 483 648 Integer.MIN_VALUE	2 147 483 647 Integer.MAX_VALUE
long	8	- 9 223 372 036 854 775 808 Long.MIN_VALUE	9 223 372 036 854 775 807 Long.MAX_VALUE

Par défaut: une constante entière est de type int.

3.4 TYPE REEL

Le type réel permet de représenter de façon approchée une partie des nombres réels.
Ainsi, pour tester l'égalité de 2 nombres réels, il est préférable de comparer la valeur absolue de leur différence à un nombre très petit.

Type	Taille (octet)	Précision (chiffres significatifs)	Valeur minimale (absolue)	Valeur maximale (absolue)
float	4	7	1.402E-45 Float.MIN_VALUE	3.402E38 Float.MAX_VALUE
double	8	15	4.94E-324 Double.MIN_VALUE	1.79E308 Double.MAX_VALUE

Par défaut: une constante réelle est de type double.

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

```
import java.awt.*;

public class EcritureReel
{

  public static void main(String[] args)
  {
    double x = 10.123456789;

    System.out.println("x= " + x);
    // au moins 1 chiffre à gauche du point décimal
    // 2 chiffres exactement à droite du point décimal
    DecimalFormat deuxDecimal = new DecimalFormat("0.00");
    System.out.println("x= " + deuxDecimal.format(x));
  }
}
```

Exécution
x= 10.123456789
x= 10,12

3.5 TYPE CARACTERE

Le caractère en Java est représenté en mémoire sur 2 octets en utilisant le code Unicode.

Déclaration d'une variable caractère
char c

Une constante caractère est notée entre apostrophe.
Exemple: 'a'

3.6 INITIALISATION

3.7 CONSTANTE

Par convention, les identificateurs de constantes sont entièrement en majuscules.
Il est toujours préférable d'utiliser des constantes symboliques plutôt que des constantes littérales.

3.8 EXPRESSION CONSTANTE

4 OPERATEURS ET EXPRESSIONS

4.1 ORIGINALITE DES NOTIONS D'OPERATEUR ET D'EXPRESSION

4.2 OPÉRATEURS ARITHMETIQUES

4.2.1 Opérateurs unaires

4.2.2 Opérateurs binaires

4.2.3 Priorité des opérateurs

4.3 CONVERSIONS IMPLICITES DANS LES EXPRESSIONS

Une expression mixte est une expression avec des opérandes de types différents.

Hiérarchie des conversions d'ajustement de type

int → long → double

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

char → int

Exemple

```
char c;
c + 1; // Résulat valeur numérique de c + 1
```

4.4 OPERATEURS RELATIONNELS (COMPARAISON)

Les opérateurs relationnels sont: <; <=, >, >=, == (égal) et != (différent).

La comparaison des caractères est basée sur le code Unicode avec les relations suivantes
'0' < '1' < … < '9' < 'A' < 'B' < … < 'Z' < 'a' < 'b' < … < 'z'

Les opérateurs == et != peuvent s'appliquer à des valeurs booléennes.
Exemple: (a < b) == (c < d)
Cette expression est vraie si les 2 comparaisons (a < b) et (c < d) sont soit toutes les 2 vraies soit toutes les 2 fausses.

4.5 OPERATEURS LOGIQUES

4.5.1 Opérateur unaire

4.5.2 Opérateurs binaires

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

Avec les 2 opérateurs court-circuit && et ||, le 2ème 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 où une condition réalise un test simultanément sur l'indice et les éléments du tableau.
Exemple: Soit un tableau t avec max éléments.
La structure conditionnelle if ((i < max) && (t[i++] == 0)) n'entraîne pas d'erreur d'exécution. En effet, si (i < max) alors t[i] existe. Si (i = max) alors t[i] n'existe plus. Ainsi, dans une telle condition avec court-circuit, quand (i = max), t[i] n'est pas testé. Cette structure conditionnelle ne pose pas de problème, contrairement à if ((i < max) & (t[i++] == 0)).

Avec l'opérateur &&, la condition dépendante doit être placée après l'autre condition.
Dans les expressions utilisant l'opérateur && 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 l'opérateur ||, il faut placer à gauche la condition susceptible d'être vraie.

4.6 OPERATEUR D'AFFECTATION

L'opérateur d'affectation impose que son 1er 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

char → int → long → double

4.7 OPERATEURS D'INCREMENTATION ET DE DECREMENTATION

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 lorsqu'il est placé à gauche de son opérande
- un opérateur de post-incrémentation lorsqu'il est placé à droite de son opérande

L'opérateur de décrémentation - - est
- un opérateur de prédécrémentation lorsqu'il est placé à gauche de son opérande
- un opérateur de postdécrémentation lorsqu'il est placé à droite de son opérande

Exemples
y = x++; équivaut à { y = x; x = x + 1; }
y = ++x; équivaut à { x = x + 1; y = x; }

4.8 OPERATEURS D'AFFECTATION ELARGIE

Les opérateurs d'affectation élargie 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; équivaut à a = a + b;

Les opérateurs relationnels et logiques ne sont pas concernés par cette possibilité.

4.9 OPERATEUR DE CAST

L'opérateur de 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)

4.10 OPERATEURS DE MANIPULATION DE BITS

4.11 OPERATEUR CONDITIONNEL

L'opérateur conditionnel est un opérateur ternaire (3 opérandes) permettant de traduire
si (expression_test) alors variable = expression_1
sinon variable = expression_2
par

variable = (expression_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 sa valeur soit utilisée.
Exemple: (a > b) ? i++ : i--
Cette instruction incrémente ou décrémente la variable i selon la condition (a > b).

5 INSTRUCTIONS DE CONTROLE

5.1 INSTRUCTION if

if (expression_booléenne) instruction_1
[else instruction_2]

5.2 INSTRUCTION switch

switch (expression)
{
  case constante_1: [suite_instructions_1]
  .....
  case constante_n: [suite_instructions_n]
  [default: suite_instructions]
}

expression est une expression de type byte, short, int ou char; les expressions de type byte, short ou char ne peuvent que de simples variables (sauf avec l'opérateur cast); le type long n'est pas autorisé
constante_i est une expression constante d'un type compatible par affectation avec le type de expression
suite_instructions_i est une suite d'instructions quelconques.

Si expression vaut constante_i, suite_instructions_i est exécutée et toutes les suites d'instructions suivantes qui peuvent être hiérarchisées.
Pour sortir du switch immédiatement, les suites d'instructions doivent se terminer par un break.
Le mot clé default permet éventuellement d'exécuter une suite_instructions si aucune valeur satisfaisante n'a été rencontré auparavant.

5.3 INSTRUCTION while

while (expression_booléenne) instruction

L'instruction while répète l'instruction tant que la condition de poursuite est vraie. Le nombre de boucles n'est pas connu à l'entrée de la boucle.
La condition est testée avant chaque parcours de la boucle. Ainsi, une telle boucle peut n'être parcourue aucune fois si la condition est fausse à l'entrée de la boucle. Cette condition doit être initialisée.
Une sortie de boucle se fait toujours après un parcours complet des instructions de la boucle et non dès que la condition devient fausse.

Soit le programme partiel suivant

```
...
int compteur = 1;
while (compteur <= 10)
  {
    Instruction_unique;
    compteur++;
  }
...
```

Compteur: 1, 2, …, 10

Un programme équivalent concis

```
...
int compteur = 0;
while (++compteur <= 10) Instruction_unique;

...
```

Compteur: 1, 2, …, 10

Mais, le programme suivant n'est pas équivalent

```
...
int compteur = 0;
while (compteur++ <= 10) Instruction_unique;

...
```

Compteur: 1, 2, …, 10, 11

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. Le nombre de boucles n'est pas connu à l'entrée de la boucle.
La condition est testée après chaque parcours de la boucle. Ainsi, une telle boucle est parcourue au moins une fois.
Une sortie de boucle se fait toujours après un parcours complet des instructions de la boucle et non dès que la condition devient fausse.
Point-virgule après expression_booléenne.

5.5 INSTRUCTION for

for ([initialisation]; [expression_booléenne] ; [incrémentations]) instruction
initialisation est une déclaration ou une suite d'expressions quelconques séparées par des virgules; la déclaration est locale au bloc régit par l'instruction for (les variables d'initialisation n'existent plus en sortie de boucle)
incrémentations sont des suites d'expressions 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+=i)
     {
       System.out.println ("i= " + i + "   j= " + j);
     }
  }
}
```

Exécution

```
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;
...
```

Un programme équivalent concis

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

5.6 INSTRUCTION break

L'instruction break ne doit être utilisée que dans une instruction swith.

5.7 INSTRUCTIONS break AVEC ETIQUETTE, continue, continue AVEC ETIQUETTE

Ces 3 instructions ne doivent pas être utilisées.

6 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

(i) Définition des champs

(ii) Définition des méthodes

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 pas 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 de cet emplacement.

Ainsi, l'affectation

uneEquation = new Equation()

construit un objet de type Equation et place sa référence dans uneEquation qui doit avoir été déclaré 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()

L'appel d'une méthode se fait donc 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 parenthèses (permettant la distinction avec un champ).

6.1.3 Plusieurs classes dans un même fichier source

Problème: Résolution d'une équation du second 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; // Coefficient du terme en X2
  private double coeffX1; // Coefficient du terme en X1
  private double coeffX0; // Coefficient du terme en X0
  private double racine1; // Première racine
  private double racine2; // Seconde racine

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

  // Méthode de résolution
  public void résolution()
  {
    double discri;

    discri = (coeffX1*coeffX1 - 4*coeffX2*coeffX0);
    racine1 = (-coeffX1 + Math.sqrt(discri)) / (2*coeffX2);
    racine2 = (-coeffX1 - Math.sqrt(discri)) / (2*coeffX2);
  }
```

```
  // 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(); // création de l'objet uneEquation

    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.text.DecimalFormat;

// Classe Equation
public class Equation
{
...
}
```

De préférence, mettre l'attribut d'accès de la classe Equation public pour des utilisations par des méthodes à l'extérieur des classes du paquetage.

(2) Compiler le fichier Equation.java
Génération d'une fichier Equation.class

(3) Sauvegarder la classe EquationSecondDegre dans un fichier EquationSecondDegre.java

(4) Compiler le fichier EquationSecondDegre.java
Il faut que le fichier Equation.class existe et qu'il soit accessible (pas de problème en utilisant le même paquetage).

6.2 CONSTRUCTEURS

6.2.1 Principe

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

Transformation de la méthode d'initialisation de la classe Equation

```
// Méthode
public void initialisation(double X2, double X1, double X0)
{
  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 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);
```

6.2.2 Quelques règles

6.2.3 Création d'un objet

6.2.4 Initialisation avec les champs d'un objet

```
class Equation
{
  // Champs
  public double coeffX2, coeffX1, coeffX0;
...
}

public class EquationSecondDegre
{
  public static void main (String[] args)
  {
    Equation uneEquation = new Equation();

    uneEquation.coeffX2 = 1.0;
    uneEquation.coeffX1 = 5.0;
    uneEquation.coeffX0 = 1.0;
...
  }
}
```

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 d'un point

nomObjet.nomChamp

Cette approche est fortement déconseillée car elle ne respecte pas le principe d'encapsulation des données.
A utiliser éventuellement avec les listes

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, c'est-à-dire des valeurs de certains champs sans les modifier

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

  public double getRacine2()
  {
    return racine2;
  }

...
}

public class EquationSecondDegre
{
  public static void main (String[] args)
  {
    ...
    System.out.println("Sol1= " + uneEquation.getRacine1());
    System.out.println("Sol2= " + uneEquation.getRacine2());
  }

}
```

Exécution

Sol1= -0.20871215252208009
Sol2= -4.7912878474779195

- les méthodes d'altération (mutator) de la forme setXXX qui modifient l'état d'un objet, c'est-à-dire les valeurs de certains champs

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

  public void setRacine2(double rac2)
  {
    racine2 = rac2;
  }
...
}

public class EquationSecondDegre
{
  public static void main (String[] args)
  {
    ...
    uneEquation.affichage();
    uneEquation.setRacine1(1.0);
    uneEquation.setRacine2(2.0);
    uneEquation.affichage();
  }
}
```

Exécution
Racine1= -0,21
Racine2= -4,79
Racine1= 1,00
Racine2= 2,00

6.4 AFFECTATION ET COMPARAISON D'OBJETS

Soient 2 variables E1 et E2 de type Equation
```
Equation E1, E2;
```
Soit 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);
```
L'instruction

E1 = E2

affecte à E1 la référence de E2.
Ainsi, E1 et E2 désigne le même objet Equation(2.0, 6.0, 2.0) et non pas 2 objets de même valeur.

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 dispose en Pascal. Java utilise un mécanisme de gestion automatique de la mémoire.

Possibilité de créer un objet sans référence
Avec la 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

```
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 DES 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.

```
public double somme()
{
  double som = 0.0;

  som = racine1 + racine2;

  return som;
}
```

Méthode appelée

```
public static void main (String[] args)
{
  ...
  System.out.println("Somme= " + uneEquation.somme());
  ...
}
```

Possibilité d'utiliser le résultat d'une méthode fonction dans une expression d'une méthode d'une autre classe

```
public static void main (String[] args)
{
  double sommeCarre = 0.0;

  ...
  sommeCarre = uneEquation.somme() * uneEquation.somme();
  System.out.println("sommeCarre= " + sommeCarre);
  ...
}
```

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

```
class Equation
{
  ...
  private double sommeSquare = 0.0;

  ...
  public double somme()
  {
    ...
  }

  public void affichage()
  {
    ...
    sommeSquare = somme() * somme();
    System.out.println("sommeSquare= " + sommeSquare);
  }

}
```

6.6.3 Arguments formels et effectifs

6.6.4 Variables locales

6.7 CHAMPS ET METHODES STATIQUES

6.7.1 Champs statique

Problème: Champ statique pour compter le nombre de résolutions

```
class Equation
{
  ...
  private static int nbResolution = 0;

  public Equation(double X2, double X1, double X0)
  {
    ...
    nbResolution++;
  }

  public void résolution()
  {
    ...
  }

  public void affichage()
  {
    System.out.println("nbResolution= " + nbResolution);
   ...
  }
}

public class EquationSecondDegre
{

  public static void main (String[] args)
  {
    Equation uneEquation = new Equation(1.0, 5.0, 1.0);
    uneEquation.résolution();
    uneEquation.affichage();

    Equation uneEquation1 = new Equation(2.0, 6.0, 2.0);
    uneEquation1.résolution();
    uneEquation1.affichage();
  }
}
```

Exécution

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

Même programme sans l'attribut static

```
class Equation
{
  ...
  private int nbResolution = 0;
  ...
}

public class EquationSecondDegre
{
   ...
}
```

Exécution

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

6.7.2 Méthodes statiques

Problème: Méthode statique pour compter le nombre de résolutions

```
class Equation
{
  ...
  private static int nbResolution = 0;

  public Equation(double X2, double X1, double X0)
  {
    ...
    nbResolution++;
  }

  public void résolution()
  {
    ...
  }

  public static int nbSolution()
  {
    return nbResolution;
  }

  public void affichage()
  {
   ...
  }
}
```

```
public class EquationSecondDegre
{

  public static void main (String[] args)
  {
    Equation uneEquation = new Equation(1.0, 5.0, 1.0);
    uneEquation.résolution();
    uneEquation.affichage();
    System.out.println("nbSolution= "
                       +  Equation.nbSolution());

    Equation uneEquation1 = new Equation(2.0, 6.0, 2.0);
    uneEquation1.résolution();
    uneEquation1.affichage();
    System.out.println("nbSolution= "
                       +  Equation.nbSolution());
  }
}
```

Exécution

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

6.7.3 Initialisation des champs statiques

Bloc d'initialisation statique

```
static
{
  ...
}
```

6.8 SURDEFINITION (SURCHARGE) DE METHODES

6.8.1 Définition

6.8.2 Surdéfinition de méthodes

```
class Equation
{
  private double coeffX2, coeffX1, coeffX0, racine1, racine2;
  private static int nbResolution = 0;

  // Constructeur vide
  public Equation()
  {
    coeffX2 = 0.0;
    coeffX1 = 0.0;
    coeffX0 = 0.0;
    nbResolution++;
  }

  // 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 X0)
  {
    coeffX2 = X2;
    coeffX1 = 0.0;
    coeffX0 = X0;
    nbResolution++;
  }
```

```
// Méthode de résolution pour 3 arguments pour ax2 + bx + c
public void résolution()
{
  double discri;

  discri = (coeffX1*coeffX1 - 4*coeffX2*coeffX0);
  racine1 = (-coeffX1 + Math.sqrt(discri)) / (2*coeffX2);
  racine2 = (-coeffX1 - Math.sqrt(discri)) / (2*coeffX2);
}

// Méthode de résolution pour 3 arguments pour ax2 + bx + c
public void résolution(double coeffX2, double coeffX1, double coeffX0)
{
  double discri;

  discri = (coeffX1*coeffX1 - 4*coeffX2*coeffX0);
  racine1 = (-coeffX1 + Math.sqrt(discri)) / (2*coeffX2);
  racine2 = (-coeffX1 - Math.sqrt(discri)) / (2*coeffX2);
}

// Méthode de résolution pour 2 arguments pour ax2 + c
public void résolution(double coeffX2, double coeffX0)
{
  double discri;

  discri = (coeffX1*coeffX1 - 4*coeffX2*coeffX0);
  racine1 = (-coeffX1 + Math.sqrt(discri)) / (2*coeffX2);
  racine2 = (-coeffX1 - Math.sqrt(discri)) / (2*coeffX2);
}

// Méthode de résolution
public static int nbSystème()
{
  return nbResolution;
}
```

```
  // 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 = new Equation();
    uneEquation.résolution(1.0, 5.0, 1.0);
    uneEquation.affichage();
    System.out.println("--------------------");

    Equation uneEquation1 = new Equation();
    uneEquation1.résolution(1.0, -5.0);
    uneEquation1.affichage();
    System.out.println("--------------------");

    System.out.println("Nombre de systèmes= " + Equation.nbSystème());
  }
}
```

Exécution

```
Racine1= -0,21
Racine2= -4,79
--------------------
Racine1= 2,24
Racine2= -2,24
--------------------
Nombre de systèmes= 2
```

6.8.3 Surdéfinition de constructeurs

```
class Equation
{
  ...
}

public class EquationSecondDegre
{

  public static void main (String[] args)
  {
    Equation uneEquation = new Equation(1.0, 5.0, 1.0);
    uneEquation.résolution();
    uneEquation.affichage();
    System.out.println("---------------------");

    Equation uneEquation1 = new Equation(1.0, -5.0);
    uneEquation1.résolution();
    uneEquation1.affichage();
    System.out.println("---------------------");

    System.out.println("Nombre de systèmes= "
                       + Equation.nbSystème());
  }

}
```

Même exécution

6.9 TRANSMISSION D'INFORMATION AVEC LES METHODES

6.9.1 Transmission par valeur

- 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 de type primitif par 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
- retourner un tableau de type primitif pour généraliser le retour d'une variable de type primitif.

6.9.2 Transmission d'objet en argument

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

```
class Equation
{
  ...

  // Méthode de test des coefficients de 2 systèmes
  // Méthode statique
  public static double testCoefficient(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;
  }
  ...
}
```

```
public class EquationSecondDegre
{

  public static void main (String[] args)
  {
    Equation uneEquation = new Equation(1.0, 5.0, 1.0);
    uneEquation.résolution();
    uneEquation.affichage();
    System.out.println("---------------------");

    Equation uneEquation1 = new Equation(2.0, 10.0, 2.0);
    uneEquation1.résolution();
    uneEquation1.affichage();
    System.out.println("---------------------");

    System.out.println("Nombre de systèmes= " + Equation.nbSystème());
    System.out.println("Facteur multiplicatif des coefficients des 2 systèmes= "
                       + Equation.testCoefficient(uneEquation, uneEquation1));
  }
}
```

Exécution

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

6.9.3 Transmission par valeur de types primitifs

1ère version

```
class Equation
{
  ...

  // Méthode de doublement des coefficients de ax2 + bc + c
  public void doublementCoefficient(double a, double b, double c)
  {
    a = 2 * a;
    b = 2 * b;
    c = 2 * c;
  }

  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));
  }
  ...
}
```

```
public class EquationSecondDegre
{

  public static void main (String[] args)
  {
    double cX2 = 1.0; // Coefficient du terme en X2
    double cX1 = 5.0; // Coefficient du terme en X1
    double cX0 = 1.0; // Coefficient du terme en X0

    Equation uneEquation = new Equation(cX2, cX1, cX0);
    uneEquation.résolution();
    uneEquation.affichage();
    System.out.println("---------------------");

    uneEquation.doublementCoefficient(cX2, cX1, cX0); // Transmission par valeur
                                                      // cX2, cX1, cX0 ne sont pas pas
                                                      // doublées

    Equation uneEquation1 = new Equation(cX2, cX1, cX0);
    uneEquation1.résolution();
    uneEquation1.affichage();
    System.out.println("---------------------");

    System.out.println("Nombre de systèmes= " + Equation.nbSystème());

  }

}
```

Exécution

```
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79
---------------------
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79
---------------------
Nombre de systèmes= 2
```

2ème version

```
class Equation
{
  ...

  public double doublementCoefficient(double a, double b, double c)
  {
    a = 2 * a;
    b = 2 * b;
    c = 2 * c;
    System.out.println("a=" + a + " b=" + b + " c=" + c);

    return a;
  }

  ...
}

public class EquationSecondDegre
{
  ...
}
```

Exécution

```
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79
----------------------
a=2.0 b=10.0 c=2.0
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79
----------------------
Nombre de systèmes= 2
```

6.9.4 Transmission par adresse de la référence d'un objet

1ère version avec un seul objet modifié

```
class Equation
{
  ...

  public void doublementCoefficient()
  {
    coeffX2 = 2 * coeffX2;
    coeffX1 = 2 * coeffX1;
    coeffX0 = 2 * coeffX0;
  }

  ...

}
```

```
public class EquationSecondDegre
{

  public static void main (String[] args)
  {
    double cX2 = 1.0; // Coefficient du terme en X2
    double cX1 = 5.0; // Coefficient du terme en X1
    double cX0 = 1.0; // Coefficient du terme en X0

    Equation uneEquation = new Equation(cX2, cX1, cX0);
    uneEquation.résolution();
    uneEquation.affichage();
    System.out.println("--------------------");

    uneEquation.doublementCoefficient();

    uneEquation.résolution();
    uneEquation.affichage();
    System.out.println("--------------------");

    System.out.println("Nombre de systèmes= " + Equation.nbSystème());

  }

}
```

Exécution

```
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79
--------------------
Equation= 2,00x2 + 10,00x + 2,00
Racine1= -0,21
Racine2= -4,79
--------------------
Nombre de systèmes= 1
```

2ème version avec 2 objets

```
class Equation
{
  ...

  public Equation doublementCoefficient()
  {
    Equation E = new Equation();

    E.coeffX2 = 2 * coeffX2;
    E.coeffX1 = 2 * coeffX1;
    E.coeffX0 = 2 * coeffX0;

    return E;
  }
  ...

}
```

```
public class EquationSecondDegre
{

  public static void main (String[] args)
  {
    double cX2 = 1.0; // Coefficient du terme en X2
    double cX1 = 5.0; // Coefficient du terme en X1
    double cX0 = 1.0; // Coefficient du terme en X0

    Equation uneEquation = new Equation(cX2, cX1, cX0);
    uneEquation.résolution();
    uneEquation.affichage();
    System.out.println("--------------------");

    Equation uneEquation1 = uneEquation.doublementCoefficient();

    uneEquation1.résolution();
    uneEquation1.affichage();
    System.out.println("--------------------");

    System.out.println("Nombre de systèmes= " + Equation.nbSystème());

    System.out.println("Facteur multiplicatif des coefficients des 2 systèmes= "
                       + Equation.testCoefficient(uneEquation, uneEquation1));
  }
}
```

Exécution

```
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79
--------------------
Equation= 2,00x2 + 10,00x + 2,00
Racine1= -0,21
Racine2= -4,79
--------------------
Nombre de systèmes= 2
Facteur multiplicatif des coefficients des 2 systèmes= 2.0
```

6.9.5 Valeur de retour d'une méthode

6.9.6 Autoréférence this

3 propriétés
(i) Le mot-clé this est une référence à l'objet courant dans sa globalité
(ia) this remplace le nom de l'instance courante.

Rappel du programme avec la méthode retournant un objet

```
class Equation
{
  ...

  public Equation doublementCoefficient()
  {
    Equation E = new Equation();

    E.coeffX2 = 2 * coeffX2;
    E.coeffX1 = 2 * coeffX1;
    E.coeffX0 = 2 * coeffX0;

    return E;
  }
  ...
}
```

Programme introduisant this artificiellement

```
class Equation
{
  ...

  public Equation doublementCoefficient()
  {
    Equation E = new Equation();

    E.coeffX2 = 2 * this.coeffX2;
    E.coeffX1 = 2 * this.coeffX1;
    E.coeffX0 = 2 * this.coeffX0;

    return E;
  }
  ...
}
```

(ib) Utilisation de noms d'arguments identiques à des noms de champs

Cette utilisation permet d'éviter de créer de nouveaux identificateurs.

Rappel du programme avec le constructeur à 3 arguments

```
class Equation
{
  ...
  public Equation(double X2, double X1, double X0)
  {
    coeffX2 = X2;
    coeffX1 = X1;
    coeffX0 = X0;
    nbResolution++;
  }

  ...
}
```

Programme avec un constructeur comportant des noms d'arguments identiques à des noms de champs

```
class Equation
{
  ...
  public Equation(double coeffX2, double coeffX1,
                  double coeffX0)
  {
    this.coeffX2 = coeffX2;
    this.coeffX1 = coeffX1;
    this.coeffX0 = coeffX0;
    nbResolution++;
  }
  ...
}
```

(iii) Appel d'un constructeur au sein d'un autre constructeur

Rappel du programme avec 3 constructeurs

```
class Equation
{
  ...

  // Constructeur vide
  public Equation()
  {
    coeffX2 = 0.0;
    coeffX1 = 0.0;
    coeffX0 = 0.0;
    nbResolution++;
  }

  // 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 X0)
  {
    coeffX2 = X2;
    coeffX1 = 0.0;
    coeffX0 = X0;
    nbResolution++;
  }

  ...

}

public class EquationSecondDegre
{

  public static void main (String[] args)
  {
    Equation uneEquation = new Equation(1.0, 5.0, 1.0);
    uneEquation.résolution();
    uneEquation.affichage();
    System.out.println("---------------------");

    Equation uneEquation1 = new Equation(1.0, -5.0);
    uneEquation1.résolution();
    uneEquation1.affichage();
    System.out.println("---------------------");

    System.out.println("Nombre de systèmes= "
                       + Equation.nbSystème());
  }
}
```

Exécution

```
Racine1= -0,21
Racine2= -4,79
---------------------
Racine1= 2,24
Racine2= -2,24
---------------------
Nombre de systèmes= 2
```

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 X0)
  {
    this(X2, 0.0, X0);
  }

  ...
}

public class EquationSecondDegre
{
  ...
}
```

Même exécution

6.10 RECURSIVITE DES METHODES

Programme factoriel

```
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= 3628800
```

6.11 CLASSES INTERNES

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

  // Champs et méthodes de la classe externe
}
```

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 méthodeExterne()
  {
    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
  }

  // Champs et méthodes de la classe externe
  // Champ de la classe externe
  private Interne i;
  ...

}
```

Rappel: Résolution d'une équation du second degré

```
class EquationClasseInterne
{

  private double coeffX2, coeffX1, coeffX0, racine1, racine2;

  public EquationClasseInterne(double coeffX2, double coeffX1, double coeffX0)
  {
    this.coeffX2 = coeffX2;
    this.coeffX1 = coeffX1;
    this.coeffX0 = coeffX0;
  }

  public void résolution()
  {
    double discri;

    discri = (coeffX1*coeffX1 - 4*coeffX2*coeffX0);
    racine1 = (-coeffX1 + Math.sqrt(discri)) / (2*coeffX2);
    racine2 = (-coeffX1 - Math.sqrt(discri)) / (2*coeffX2);
  }

  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));
  }
}
```

```
public class EquationSecondDegre
{
  public static void main (String[] args)
  {
    Equation uneEquation = new Equation(1.0, 5.0, 1.0);
    uneEquation.résolution();
    uneEquation.affichage();
    System.out.println("---------------------");

    Equation uneEquation1 = new Equation(0.0, 10.0, -10.0);
    uneEquation1.résolution();
    uneEquation1.affichage();
  }
}
```

Exécution

```
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79
---------------------
Equation= 0,00x2 + 10,00x + -10,00
Racine1= ?
Racine2= -?
```

Problème: Résolution d'une équation du second degré avec le cas particulier d'une équation du premier degré

```
class Equation
{

  class EquationPremierDegre
  {
    private int nbEquationPremierDegre = 0;

    // Méthode de résolution de bx + c = 0
    public void résolution()
    {
      System.out.println("Détermination de la solution");
      racine1 = -coeffX0 / coeffX1;
      racine2 = Double.NaN;
      nbEquationPremierDegre++;
    }
  }

  // Champs
  private double coeffX2, coeffX1, coeffX0, racine1, racine2;

  // Constructeur à 3 arguments pour ax2 + bx + c
  public Equation(double coeffX2, double coeffX1, double coeffX0)
  {
    this.coeffX2 = coeffX2;
    this.coeffX1 = coeffX1;
    this.coeffX0 = coeffX0;
  }
```

```
  // Méthode de résolution pour 3 arguments pour ax2 + bx + c = 0
  public void résolution()
  {
    double discri;

    if (coeffX2 != 0.0)
    {
      discri = (coeffX1*coeffX1 - 4*coeffX2*coeffX0);
      racine1 = (-coeffX1 + Math.sqrt(discri)) / (2*coeffX2);
      racine2 = (-coeffX1 - Math.sqrt(discri)) / (2*coeffX2);
    }
    else
    {
      System.out.println("Traitement d'une équation du premier degré");
      EquationPremierDegre E1 = new EquationPremierDegre();
      E1.résolution();
      System.out.println("nbEquationPremierDegre= " + E1.nbEquationPremierDegre);
    }
  }

  // Méthode d'affichage
  public void affichage()
  {
    ...
  }
}

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

Exécution

```
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79
----------------------
Traitement d'une équation du premier degré
Détermination de la solution
nbEquationPremierDegre= 1
Equation= 0,00x2 + 10,00x + -10,00
Racine1= 1,00
Racine2= ?
```

6.12 PAQUETAGES

6.12.1 Définition

6.12.2 Attribution d'une classe à un paquetage

6.12.3 Utilisation d'une classe d'un paquetage

(i) Donner le nom du paquetage avec le nom de la classe

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

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

(ii) Importer la classe

```
import coursjava.Equation;

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

(iii) Importer le paquetage

```
import coursjava.*;

public class EquationSecondDegre
{

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

6.12.4 Paquetages standard

6.12.5 Portée des classes

6.12.6 Portée des champs et des méthodes

7 TABLEAUX

7.1 DECLARATION ET CREATION DE TABLEAUX

7.1.1 Introduction

7.1.2 Déclaration d'un tableau

7.1.3 Création d'un tableau

(i) Création par l'opérateur new
(ii) Création par initialisation

7.2 UTILISATION DE TABLEAUX

7.2.1 Accès individuel aux éléments d'un tableau

7.2.2 Accès global au tableau (affectation de références)

7.2.3 Taille d'un tableau

7.3 TABLEAU D'OBJETS

Programme classique sans tableau

```
public class EquationSecondDegre
{

  public static void main (String[] args)
  {
    Equation uneEquation = new Equation(1.0, 5.0, 1.0);
    uneEquation.résolution();
    uneEquation.affichage();
    System.out.println("---------------------");

    Equation uneEquation1 = new Equation(1.0, 10.0, 1.0);

    uneEquation1.résolution();
    uneEquation1.affichage();
    System.out.println("---------------------");

    System.out.println("Nombre de systèmes= "
                       + Equation.nbSystème());
  }
}
```

Exécution

```
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79
---------------------
Equation= 1,00x2 + 10,00x + 1,00
Racine1= -0,10
Racine2= -9,90
---------------------
Nombre de systèmes= 2
```

Programme avec tableau

```
public class EquationSecondDegre
{

  public static void main (String[] args)
  {
    Equation[] système = 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 < système.length; i++)
    {
      système[i].résolution();
      système[i].affichage();
      System.out.println("----------------------");
    }

    System.out.println("Nombre de systèmes= "
                       + Equation.nbSystème());
  }

}
```

Même exécution

7.4 TABLEAU EN ARGUMENT

Programme tableau en argument

```
class Equation
{
  ...

  // Méthode de doublement des coefficients ax2 + bc + c
  public static void doublementCoefficient(double coeff[])
  {
    for (int i = 0; i < coeff.length; i++) coeff[i] = 2 * coeff[i];
  }
   ...
}

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

    système[0] = new Equation(coeff[0], coeff[1], coeff[2]);

    Equation.doublementCoefficient(coeff);

    système[1] = new Equation(coeff[0], coeff[1], coeff[2]);

    for (int i = 0; i < système.length; i++)
    {
      système[i].résolution();
      système[i].affichage();
      System.out.println("---------------------");
    }

    System.out.println("Nombre de systèmes= " + Equation.nbSystème());
  }
}
```

Exécution

```
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79
----------
Equation= 2,00x2 + 10,00x + 2,00
Racine1= -0,21
Racine2= -4,79
----------
Nombre de systèmes= 2
```

7.5 TABLEAUX MULTIDIMENSIONNELS

8 HERITAGE

8.1 INTRODUCTION

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

Classe de base sans constructeur et sans classe dérivée

```
class Equation
{
  private double coeffX2, coeffX1, coeffX0, racine1,
                 racine2;

  public void init(double coeffX2, double coeffX1,
                   double coeffX0)
  {
    this.coeffX2 = coeffX2;
    this.coeffX1 = coeffX1;
    this.coeffX0 = coeffX0;
  }

  public void résolution()
  {
    ...
  }

  public void affichage()
  {
    ...
  }
}

public class EquationSecondDegre
{
  public static void main(String[] args)
  {
    Equation uneEquation = new Equation();

    uneEquation.init(1.0, 5.0, 1.0);
    uneEquation.résolution();
    uneEquation.affichage();
    System.out.println("---------------------");
  }
}
```

Exécution

```
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79
---------------------
```

Classe de base sans constructeur et classe dérivée sans constructeur

```
// Classe Equation
class Equation
{
  ...
}

// Classe Dérivée
class ClasseDerivée extends Equation
{
  public void initCD(double a, double b, double c)
  {
    init(a, b, c);
  }
}

public class EquationSecondDegre
{
  public static void main(String[] args)
  {
    ClasseDerivée cD = new ClasseDerivée();

    cD.initCD(1.0, 5.0, 1.0);
    cD.résolution();  // ACCES A LA METHODE PUBLIQUE
                         DE LA CLASSE DE BASE
    cD.affichage();  // ACCES A LA METHODE PUBLIQUE
                        DE LA CLASSE DE BASE
    System.out.println("---------------------");
  }
}
```

Même exécution

Classe de base sans constructeur et classe dérivée avec constructeur

```
class Equation
{
  ...
}

class ClasseDerivée extends Equation
{
  public ClasseDerivée(double a, double b, double c)
  {
    init(a, b, c);
  }
}

public class EquationSecondDegre
{
  public static void main(String[] args)
  {
    ClasseDerivée cD = new ClasseDerivée(1.0, 5.0, 1.0);

    cD.résolution();
    cD.affichage();
    System.out.println("---------------------");
  }
}
```

Même exécution

8.3 CONSTRUCTION ET INITIALISATION DES OBJETS DERIVES

8.3.1 Appel du constructeur

```
class ClasseDerivée extends Equation
{
  public ClasseDerivée(double a, double b, double c)
  {
    super(a, b, c);
  }
}
```

Classe de base avec constructeur et classe dérivée avec un constructeur

```
class Equation
{
  ...

  public Equation(double coeffX2, double coeffX1,
                  double coeffX0)
  {
    this.coeffX2 = coeffX2;
    this.coeffX1 = coeffX1;
    this.coeffX0 = coeffX0;
  }

  ...
}

class ClasseDerivée extends Equation
{
  public ClasseDerivée(double a, double b, double c)
  {
    super(a, b, c);
  }
}

public class EquationSecondDegre
{
  public static void main (String[] args)
  {
    ClasseDerivée cD = new ClasseDerivée(1.0, 5.0, 1.0);

    cD.résolution();
    cD.affichage();
    System.out.println("---------------------");
  }
}
```

Même exécution

Classe de base sans constructeur

```
class A
{
  // Pas de constructeur
  ...
}

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

Classe dérivée sans constructeur

- soit le constructeur par défaut de la classe de base

```
class A
{
  // Pas de constructeur
  ...
}

class B extends A
{
  // Pas de constructeur
  ...
}
```

- soit le constructeur public vide de la classe de base.

```
class A
{
  public A()
  {
    ...
  }

  public A(int n)
  {
    ...
  }

  ...
}

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

```
class A
{
  public A(int n)
  {
    ...
  }

  ...
}

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

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

8.3.2 Initialisation d'un objet dérivé

8.4 DERIVATIONS SUCCESSIVES

```
class Point
{
  protected int x, y;

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

  public Point()
  {
    this(0, 0);
  }
}

class Cercle extends Point
{
  private double r;

  public Cercle(int x, int y, double r)
  {
    super(x, y);
    this.r = (r > 0.0 ? r : 0.0);
  }

  public Cercle()
  {
    this(0, 0, 0.0);
  }
}
```

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

    p1 = c; // Affecte la référence de la classe dérivée
              Cercle à la référence de la superclasse Point
    c1 = (Cercle)p1; // Remplacement de la référence de la
                        superclasse Point par la référence
                        de la classe dérivée Cercle
    c1 = (Cercle)p; // Erreur d'exécution
    if (p instanceof Cercle) c1 = (Cercle)p;
    else System.out.printeln("p ne référence pas un cercle");
  }
}
```

8.5 REDEFINITION ET SURDEFINITION DE MEMBRES

8.5.1 Redéfinition de méthodes

Classe dérivée avec une redéfinition de méthode

```
class Equation
{
  ...

  public void affichage()
  {
    ...
  }
}

class ClasseDerivée extends Equation
{
  public ClasseDerivée(double a, double b, double c)
  {
    super(a, b, c);
  }

  public void affichage()
  {
    System.out.println("Equation du second degré");
  }
}

public class EquationSecondDegre
{
  public static void main(String[] args)
  {
    ClasseDerivée cD = new ClasseDerivée(1.0, 5.0, 1.0);

    cD.résolution();
    cD.affichage();
    System.out.println("---------------------");
  }
}
```

Exécution
Equation du second degré

```
class Equation
{
  ...
}

class ClasseDerivée extends Equation
{

  ...

  public void affichage()
  {
    System.out.println("Equation du second degré");
    super.affichage();
  }
}

public class EquationSecondDegre
{
  ...
}
```

Exécution
Equation du second degré
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79

8.5.2 Surdéfinition (surcharges) 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); erreur 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

8.5.4 Contraintes portant sur la redéfinition

La redéfinition d'une méthode ne doit pas diminuer les droits d'accès à cette méthode. Elle peut les augmenter, c'est-à-dire étendre sa zone de visibilité.

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

class B extends A
{
  private void m(int x)
  {
    ...
  }
}
```

```
class A
{
  private void m(int n)
  {
    ...
  }
}

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

8.5.5 Règles générales de redéfinition et de surdéfinition

8.5.6 Duplication de champs

8.6 POLYMORPHISME

8.6.1 Définition

8.6.2 Polymorphisme et gestion d'un tableau hétérogène

```
class Equation
{
  ...

  public void affichage()
  {
    ...
  }
}

class ClasseDerivée extends Equation
{
  public void affichage()
  {
    System.out.println("Equation du second degré");
    super.affichage();
  }
}

public class EquationSecondDegre
{
  public static void main(String[] args)
  {
    Equation système[] = new Equation[2];

    système[0] = new Equation(1.0, 5.0, 1.0);
    système[1] = new ClasseDerivée(2.0, 10.0, 2.0);

    for (int i = 0; i < système.length; i++)
    {
      système[i].résolution();
      système[i].affichage();
      System.out.println("---------------------");
    }
  }
}
```

Exécution

```
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79
---------------------
Equation du second degré
Equation= 2,00x2 + 10,00x + 2,00
Racine1= -0,21
Racine2= -4,79
---------------------
```

8.6.3 Polymorphisme et absence de méthode dans une classe dérivée

```
class Equation
{
   ...

  public void affichage()
  {
     ...
  }
}

class ClasseDerivée extends Equation
{
//  public void affichage()
//  {
//    System.out.println("Equation du second degré");
//    super.affichage();
//  }
}

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

Exécution

```
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79
---------------------
Equation= 2,00x2 + 10,00x + 2,00
Racine1= -0,21
Racine2= -4,79
---------------------
```

8.6.4 Polymorphisme et structuration des objets

```
class Equation
{
  ...

  public void affichageCommun()
  {
    affichageSpécifique();
    ...
  }

  public void affichageSpécifique()
  {
    System.out.println("Equation du second degré");
  }
}

class ClasseDerivée extends Equation
{
  public ClasseDerivée(double a, double b, double c)
  {
    super(a, b, c);
  }

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

```
public class EquationSecondDegre
{
  public static void main(String[] args)
  {
    Equation système[] = new Equation[2];

    système[0] = new Equation(1.0, 5.0, 1.0);
    système[1] = new ClasseDerivée(2.0, 10.0, 2.0);

    for (int i = 0; i < système.length; i++)
    {
      système[i].résolution();
      système[i].affichageCommun();
      System.out.println("--------------------");
    }
  }
}
```

Exécution

```
Equation du second degré
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79
---------------------
Equation du second degré de la classe dérivée
Equation= 2,00x2 + 10,00x + 2,00
Racine1= -0,21
Racine2= -4,79
---------------------
```

8.6.5 Polymorphisme et surdéfinition

8.6.6 Règles du polymorphisme

8.6.7 Opérateur instanceof

L'opérateur binaire instanceof prend un objet comme 1er opérande et une classe comme 2ème 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

8.7 SUPER CLASSE OBJET

8.7.1 Définition

```
class Equation
{
  ...
}
```

est équivalent

```
class Equation extends Object
{
  ...
}
```

8.7.2 Utilisation d'une référence de type Object

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

o = uneEquation
```

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

o = uneEquation

((Equation)o).affichage();

// OU

Equation uneAutreEquation = (Equation)o;
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)
- l'adresse de l'objet en hexadécimal précédée de @

```
public class EquationSecondDegre
{
  public static void main(String[] args)
  {
    Equation système[] = new Equation[2];

    système[0] = new Equation(1.0, 5.0, 1.0);
    système[1] = new ClasseDerivée(2.0, 10.0, 2.0);

    System.out.println("système[0]= "
                       + système[0].toString());
    System.out.println("système[1]= "
                       + système[1].toString());
  }
}
```

Exécution

```
système[0]= coursjava.Equation@310d42
système[1]= coursjava.ClasseDerivée@5d87b2
```

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

```
class ClasseDerivée extends Equation
{
  public String toString()
  {
    return "Redéfinition de la méthode toString";
  }
}

public class EquationSecondDegre
{
  public static void main(String[] args)
  {
    Equation système[] = new Equation[2];

    système[0] = new Equation(1.0, 5.0, 1.0);
    système[1] = new ClasseDerivée(2.0, 10.0, 2.0);

    System.out.println("système[0]= "
                       + système[0].toString());
    System.out.println("système[1]= "
                       + système[1].toString());
  }
}
```

Exécution

```
système[0]= coursjava.Equation@310d42
système[1]= Redéfinition de la méthode toString
```

```
public class EquationSecondDegre
{
  public static void main(String[] args)
  {
    Equation système[] = new Equation[2];

    système[0] = new Equation(1.0, 5.0, 1.0);
    système[1] = new ClasseDerivée(2.0, 10.0, 2.0);

    System.out.println("système[0]= " + système[0]);
    System.out.println("système[1]= " + système[1]);
  }
}
```

Même exécution

8.7.4 Utilisation de la méthode equals de la classe Object

La méthode equals compare les adresses de 2 objets: o1.equals(o2).

```
public class EquationSecondDegre
{
  public static void main(String[] args)
  {
    Equation système[] = new Equation[2];

    système[0] = new Equation(1.0, 5.0, 1.0);
    système[1] = new ClasseDerivée(1.0, 5.0, 1.0);

    System.out.println("système[0]= " + système[0]);
    System.out.println("système[1]= " + système[1]);
    System.out.println("Egalité des adresses= "
                       + système[0].equals(système[1]));
  }
}
```

Exécution

```
système[0]= coursjava.Equation@310d42
système[1]= coursjava.ClasseDerivée@5d87b2
Egalité des adresses= false
```

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

```
class Equation
{
  ...

  public boolean equals(ClasseDerivée e)
  {
    return ((coeffX2 == e.coeffX2)
         && (coeffX1 == e.coeffX1)
         && (coeffX0 == e.coeffX0));
  }
}

public class EquationSecondDegre
{
  public static void main(String[] args)
  {
    Equation système[] = new Equation[2];

    système[0] = new Equation(1.0, 5.0, 1.0);
    système[1] = new ClasseDerivée(1.0, 5.0, 1.0);

    System.out.println("système[0]= " + système[0]);
    System.out.println("système[1]= " + système[1]);
    System.out.println("Egalité des coefficients= "
                       + système[0].equals(système[1]));
  }
}
```

Exécution

```
système[0]= coursjava.Equation@310d42
système[1]= coursjava.ClasseDerivée@5d87b2
Egalité des coefficients= true
```

8.7.5 Autres méthodes de la classe Object

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

8.9 CLASSES ABSTRAITES

8.9.1 Définition

```
abstract class A
{
  ...
}
```

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

```
abstract class A
{
  public void résolution()
  {
    ...
  }

  public abstract void affichage();
}
```

Déclaration

```
A a;
```

Instanciation impossible

```
a = new A(...);
```

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(...);
```

également possible

```
A a = new B(...);
```

8.9.2 Propriétés

8.9.3 Objectifs des classes abstraites

Classe abstraite pour la résolution d'équations polynomiales

```
// Super classe abstraite
abstract class EquationGénéral
{
  public abstract void résolution();
  public void affichage()
  {
    System.out.print("Equation polynomiale: ");
  }
}

// Classe Equation du second degré dérivée
class EquationDegré2 extends EquationGénéral
{
  private double a, b, c, racine1, racine2;

  public EquationDegré2(double a, double b, double c)
  {
    ...
  }

  public void résolution()
  {
    ...
  }

  public void affichage()
  {
    super.affichage();
    System.out.println("Equation quadratique");
    ...
  }
}
```

```
// Classe Equation du premier degré dérivée
class EquationDegré1 extends EquationGénéral
{
  private double a, b, solution;

  public EquationDegré1(double a, double b)
  {
    ...
  }

  public void résolution()
  {
    ...
  }

  public void affichage()
  {
    super.affichage();
    System.out.println("Equation linéaire");
    ...
  }
}

public class TestEquation
{
  public static void main(String[] args)
  {
    EquationGénéral système[] = new EquationGénéral[2];

    système[0] = new EquationDegré2(1.0, 5.0, 1.0);
    système[1] = new EquationDegré1(1.0, 5.0);

    for (int i = 0; i < système.length; i++)
    {
      système[i].résolution();
      système[i].affichage();
      System.out.println("----------------------");
    }
  }
}
```

Exécution

```
Equation polynomiale: Equation quadratique
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79
----------
Equation polynomiale: Equation linéaire
Equation= 1,00x + 5,00
Solution= -5,00
----------
```

8.10 INTERFACES

8.10.1 Introduction

L'interface possède un niveau d'abstraction supérieur à la classe abstraite avec
- des méthodes uniquement abstraites
- aucun champs, à l'exception des constantes.

8.10.2 Propriétés

8.10.3 Définition d'une interface

La définition d'une interface utilise le mot clé interface à la place de class.

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

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

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();
}
```

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

```
public interface I
{
  ...
}

class A implements I
{
  ...
}

I i = new A(...);
```

Interface pour la résolution d'équations polynomiales: transformation du programme avec classe abstraite en programme avec interface

```
// Interface
interface EquationGénéral
{
  public abstract void résolution();
  public void affichage();
}

// Classe Equation du second degré dérivée
class EquationDegré2 implements EquationGénéral
{
  ...

  public void affichage()
  {
//    super.affichage();
    System.out.print("Equation polynomiale: ");
    System.out.println("Equation quadratique");
    ...
  }
}

// Classe Equation du premier degré
class EquationDegré1 implements EquationGénéral
{
  ...

  public void affichage()
  {
//    super.affichage();
    System.out.print("Equation polynomiale: ");
    System.out.println("Equation linéaire");
    ...
  }
}

public class TestEquation
{
  public static void main(String[] args)
  {
    EquationGénéral système[] = new EquationGénéral[2];

    système[0] = new EquationDegré2(1.0, 5.0, 1.0);
    système[1] = new EquationDegré1(1.0, 5.0);

    ...
  }
}
```

Même exécution

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.

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

class A
{
  ...
}

class B extends A implements I
{
  // Redéfinition de résolution();
  // Redéfinition de affichage();
}
```

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

public interface I2
{
  void affichage();
}

class A implements I1
{
  // Redéfinition de résolution();
}

class B extends A implements I2
{
  // Redéfinition de affichage();
}
```

8.10.7 Interface et constante

```
// Interface
interface EquationGénéral
{
  static final int N = 100;
  ...
}
```

```
// Interface
interface EquationGénéral
{
  int N = 100;
  ...
}
```

Ces constantes sont également accessibles en dehors de toute classe implémentant l'interface.

```
EquationGénéral.N
```

8.10.8 Dérivation d'une interface

```
interface I1
{
  static final int N = 10;
  void résolution();
}

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

Définition équivalente de I2

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

8.10.9 Conflits de noms

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

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

class A implements I1, I2
{
  ...
}
```

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

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

class A implements I1, I2
{
  ...
}
```

8.11 CLASSES ENVELOPPES

Principales classes enveloppes
Boolean, Character, Integer, Long et Double.

Constructeur d'une classe enveloppe
ClasseEnveloppe(typeSimple)

Méthode permettant de retrouver la valeur dans le type primitif
typeSimpleValue()

8.12 PRINCIPE DE CONCEPTION DES CLASSES

L'héritage construit 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.

Programme relation "est"

```
class Equation
{
  ...

  public void résolution()
  {
    ...
  }

  public void affichage()
  {
    ...
  }
}

class ClasseDerivée extends Equation // RELATION EST
{
  public ClasseDerivée(double a, double b, double c)
  {
    super(a, b, c);
  }
}

public class EquationSecondDegre
{
  public static void main (String[] args)
  {
    ClasseDerivée cD = new ClasseDerivée(1.0, 5.0, 1.0);

    cD.résolution(); // POSSIBLE CAR RELATION EST
    cD.affichage(); // POSSIBLE CAR RELATION EST
  }
}
```

Exécution

```
Equation= 1,00x2 + 5,00x + 1,00
Racine1= -0,21
Racine2= -4,79
```

La classe construit 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.

Transformation du programme relation "est" en programme relation "a"

```
class Equation
{
  ...
}

class ClasseDerivée
{
  public Equation E; // RELATION A MAIS REGLE
                     // ENCAPSULATION NON RESPECTEE

  public ClasseDerivée (double a, double b, double c)
  {
//    super(a, b, c); IMPOSSIBLE
   E = new Equation(a, b, c);
  }
}

public class EquationSecondDegre
{
  public static void main (String[] args)
  {
    ClasseDerivée cD = new ClasseDerivée(1.0, 5.0, 1.0);

    cD.E.résolution(); // E DOIT ETRE DECLARE PUBLIC
    cD.E.affichage();
    System.out.println("---------------------");
  }
}
```

Même exécution

Transformation du programme relation "a" en respectant la règle d'encapsulation

```
class Equation
{
  ...
}

class ClasseDerivée
{
  private Equation E; // RELATION A AVEC REGLE
                      // ENCAPSULATION RESPECTEE

  public ClasseDerivée (double a, double b, double c)
  {
//    super(a, b, c); IMPOSSIBLE
   E = new Equation(a, b, c);
   E.résolution();
   E.affichage();
  }
}

public class EquationSecondDegre
{
  public static void main (String[] args)
  {
    ClasseDerivée cD = new ClasseDerivée(1.0, 5.0, 1.0);
  }
}
```

Même exécution

8.13 CLASSES ANONYMES

Une classe anonyme est une classe sans nom temporaire.

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

(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 dérivée de A
    };
```

équivalent à

```
class A1 extends A
{
  ...
}

A1 a = new A1();
```

Programme précédent avec une classe anonyme dérivée d'une classe

```
class Equation
{
  ...

  public void affichage()
  {
    ...
  }
}

class EquationModif
{
  private Equation E;

  public EquationModif(double a, double b, double c)
  {
    E = new Equation(a, b, c)
        {
          public void affichage()
          {
            System.out.println("Pas d'affichage");
          }
        };
   E.résolution();
   E.affichage();
  }
}

public class EquationSecondDegre
{
  public static void main (String[] args)
  {
    EquationModif cD = new EquationModif(1.0, 5.0, 1.0);
  }
}
```

Exécution

Pas d'affichage

(ii) Classe anonyme implémentant une interface

Programme précédent avec une classe anonyme implémentant une interface

```
class Equation
{
  ...

  public void affichage()
  {
    ...
  }
}

interface I
{
  public void affichage();
}

class EquationModif
{
  private Equation E;
  private I i;

  public EquationModif(double a, double b, double c)
  {
    E = new Equation(a, b, c);
    E.résolution();

    i = new I()
        {
          public void affichage()
          {
            System.out.println("Pas d'affichage");
          }
        };
    i.affichage();
  }
}

public class EquationSecondDegre
{
  public static void main (String[] args)
  {
    EquationModif cD = new EquationModif(1.0, 5.0, 1.0);
  }
}
```

Même exécution

9 CHAINES DE CARACTERES

9.1 CHAINES DE CARACTERES (OBJET DE TYPE STRING)

9.1.1 Introduction

9.1.2 Valeur d'un objet de type String

Un objet de type String n'est pas modifiable.

Les références à des objets de type String peuvent être modifiées.

```
chTemp = ch1;
ch1 = ch2;
ch2 = chTemp;
```

9.1.3 Entrées/sorties de chaînes

9.1.4 La méthode de longueur de chaîne length()

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

9.1.6 L'opérateur de concaténation de chaînes +

9.1.7 Conversion des opérandes de l'opérateur +

L'opérateur + peut être utilisé avec un opérande de type primitif (entier, réel, booléen) et un opérande de type chaîne. La valeur de l'opérande de type primitif est automatiquement convertie en chaîne (opération de formatage).

L'opérateur + peut être utilisé avec un opérande de n'importe quel type objet et un opérande de type chaîne. La valeur de l'objet est automatiquement convertie en chaîne grâce à l'appel de la méthode toString de la classe de l'objet qui doit être redéfinie (sinon cette méthode fournit le nom de la classe et l'adresse de l'objet).

9.1.8 L'opérateur de concaténation de chaînes +=

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()

9.3 METHODES DE COMPARAISON DE CHAINES

9.3.1 Les opérateurs == et !=

9.3.2 La méthode de comparaison de 2 chaînes equals

9.3.3 La méthode de comparaison de 2 chaînes compareTo

9.4 MODIFICATION DE CHAINES

9.4.1 La méthode de remplacement de caractères replace

9.4.2 La méthode d'extraction de sous-chaîne substring

9.4.3 La méthode de passage en majuscule ou minuscule toLowerCase et toUpperCase

9.5 TABLEAU DE CHAINES

```
public static void main(String[] args)
{
  String[] tabCh = {"Fortran", "Pl1", "Pascal", "Java"};

  for (int i = 0; i < tabCh.length; i++)
      System.out.println(tabCh[i]);
}
```

Exécution

Fortran
Pl1
Pascal
Java

```
public static void main(String[] args)
{
  String[] tabCh = {"Fortran", "Pl1", "Pascal", "Java"};

  for (int i = 0; i < tabCh.length; i++)
  {
    for (int j = 0; j < tabCh[i].length(); j++)
        System.out.print(tabCh[i].charAt(j));
    System.out.println();
  }
}
```

Même exécution

9.6 CONVERSIONS ENTRE CHAINES ET TYPES PRIMITIFS

9.6.1 Conversion d'un type, primitif ou objet, en une chaîne

public static String valueOf(Type_primitif p)
public static String valueOf(Object 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 enveloppent, l'expression

String.valueOf(n)

est équivalente à

(new Integer(n)).toString()

9.6.2 Conversion d'une chaîne en type primitif

Pour la classe enveloppe Integer

public static int parseInt(String s) throws NumberFormatException

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

9.7 CONVERSIONS ENTRE CHAINES ET TABLEAUX DE CARACTERES

9.7.1 Conversion d'un tableau de caractères en chaîne

public String(char[ ] value)

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

9.7.2 Conversion d'une chaîne en tableau de caractères

public char[ ] toCharArray()

public void getChars(int srcBegin, int srcEnd, char[ ] dst, int dstBegin)

9.8 ARGUMENTS DE LA LIGNE DE COMMANDE

En-tête de la méthode main

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

9.9 LA CLASSE StringBuffer

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, la perte de temps et la place mémoire sont importantes.

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)

Principales méthodes de la classe StringBuffer

11 FLUX

11.1 INTRODUCTION

11.2 LES FLUX TEXTE

11.2.1 Généralités

11.2.2 Ecriture d'un fichier texte

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(FileWriter fw)

PrintWriter f = new PrintWriter(fw)
ou
PrintWriter f = new PrintWriter(new FileWriter("fichier.txt"))

Fermeture d'un fichier texte en écriture
Méthode

public void close()

```
// Méthode d'affichage à l'écran
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

l**Ouverture d'un fichier texte en lecture**
Un constructeur

public FileReader(String fileName) throws FileNotFoundException

FileReader fr = new FileReader("fichier.txt")

Un constructeur

public BufferedReader(FileReader fr)

BufferedReader f = new BufferedReader(fr)

ou

BufferedReader f = new BufferedReader(new FileReader("fichier.txt"))

Fermeture d'un fichier texte en lecture
Méthode

public void close() throws IOException

```
import java.io.*;

public class LectureFichier
{
  public static void main(String[] args)
  {
    String ligne;
    BufferedReader f;

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

      while ((ligne = f.readLine()) != 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

Constructeur

public StringTokenizer(String str, String delim)

Méthodes
- **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

```
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.print("Token:" + x + " ");
        }
        System.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
```

12 LA CLASSE java.lang.Math

12.1 CHAMPS STATIQUES DE LA CLASSE java.lang.math

12.2 METHODES STATIQUES DE LA CLASSE java.lang.math

12.3 CLASSE Random DE java.util

Constructeur
- **Random()**: génération de nombres aléatoires en fonction de l'horloge interne (séquences aléatoires différentes)
- **Random(long seed)**: génération de nombres aléatoires en fonction 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 number generator's sequence
- void nextBytes(byte[ ] bytes): generates random bytes and places them into a user-supplied byte array
- **double nextDouble()**: returns the next pseudorandom, uniformly distributed double value between 0.0 and 1.0 from this random number generator's sequence
- float nextFloat(): returns the next pseudorandom, uniformly distributed float value between 0.0 and 1.0 from this random number generator's sequence
- **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(): returns the next pseudorandom, uniformly distributed int value from this random number generator's sequence
- **int nextInt(int n)**: returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence
- long nextLong(): returns the next pseudorandom, uniformly distributed long value from this random number generator's sequence
- **void setSeed(long seed)**: sets the seed of this random number generator using a single long seed

```
import java.util.*;

public class TestRandom
{
  public static void main(String[] args)
  {
    final int INITIALISATEUR = 100000;
    final int BORNESUP = 100;
    int nbAléatoire;
    Random generator = new Random(INITIALISATEUR);
    Random generator1 = new Random();// Random(System.currentTimeMillis())

    System.out.println("Série aléatoire constante");
    for (int i = 0; i < 10; i++)
    {
      nbAléatoire = generator.nextInt(BORNESUP);
      System.out.print(nbAléatoire + " ");
    }
    System.out.println("\nSérie aléatoire variable");
    for (int i = 0; i < 10; i++)
    {
      nbAléatoire = generator1.nextInt(BORNESUP);
      System.out.print(nbAléatoire + " ");
    }
  }
}
```

Exécution 1

Série aléatoire constante
87 65 9 28 45 83 91 22 48 44

Série aléatoire variable
71 99 36 48 72 42 48 21 92 77

Exécution 2

Série aléatoire constante
87 65 9 28 45 83 91 22 48 44

Série aléatoire variable
85 81 66 62 56 98 14 99 46 71