Introduction à POO et JAVA

Ecole Centrale de Lyon

2008-2009

Alexander Saidi

Novembre 2008

Plan :

1- Introduction à la POO et à JAVA

2- Programmation Java (apprentissage par l'exemple)

3- UML (Unified Modeling Language)

4- Aspects Graphiques (awt, swt, Swing, ...)

0.1 A quoi ressemble un programme Java

0.1.1 Un premier programme Java

```
1  public class Bienvenu {

3      public static void main( String args[] ) {
           System.out.println( "Bienvenu à la Programmation en Java!" );
5      }
   }
7


9  Compilation : javac Bienvenu.java
   Exécution : java Bienvenu
11 Sortie : Bienvenu à la Programmation en Java!
```

Listing 1 – un premier exemple

- Une classe "enrobe" toute fonction
- La fonction "main"

0.1.2 Quelques règles, style et conventions

✘ Tout programme Java doit contenir au moins une classe

✘ Java est sensible à la 'case' (maj/min)

✘ Les mots clefs du langage sont toujours en minuscules

➥ Un mot clef ne peut pas être un nom de variable.

✘ Par convention, le nom d'une classe commence par une majuscule.

➥ Il peut comporter des *lettres*, *chiffres*, *'_'*, *'$'*

➥ Il ne peut commencer par un chiffre et ne contient pas d'espace

◗ Par convention, un identificateur commence par une minuscule

✘ Pour la classe public *Bienvenu*, le nom du fichier doit être *Bienvenu.java*

✘ Le point d'entrée = fonction *main* (il en faut au mins une)

0.1.3 Modification de l'exemple

- Modifions l'exemple pour montrer les capacités de *println* (en fait, de *String*)

```
1 public class Parole {

3     public static void main( String args[] ) {
          System.out.println( "La raison de toute chose est " + 42 + "!!" );
5     }
  }
7
Sortie : La raison de toute chose est  + 42  !!
```

Listing 2 – Deuxième exemple

✘ Le '+' : concaténation de chaînes

✘ On peut mettre des caractères spéciaux dans une chaîne :

➥ $\backslash n, \backslash t, \backslash"$, ... (cf. langage C)

0.2 Affichage avec Printf

- Ecriture à la C avec *printf*

```
public class Avec_printf
{

   public static void main( String args[] )
   {
      System.out.printf( "%s\n%s\n",
         "Bienvenu à", " la Programmation en Java!" );
   }
}

Sortie : Bienvenu à
         la Programmation en Java!
```

Listing 3 – Sorties avec printf

✓ Format selon C/C++

➥ %s pour afficher une chaine; %d pour un entier, ...

➥ $\backslash n$ = retour à la ligne

0.3 Lecture de valeurs et Opérateuts arithmétiques

• Exemple d'addition de 2 entiers lus au clavier

```
import java.util.Scanner;
public class Addition
{  public static void main( String args[] )
   {
      Scanner input = new Scanner( System.in );
      int  val1;
      int  val2, sum;

      System.out.print( "Le premier entier : " );
      val1 = input.nextInt();
      System.out.print( "Le second entier: " );
      val2 = input.nextInt();
      sum = val1 + val2;
      System.out.printf( "La somme de %d et de %d =  %d\n", val1, val2, sum );
   }
}

Sortie : Le premier entier : 12
         Le second entier: 21
         La somme de 12 et de 21 =  33
```

Listing 4 – Utilisation d'entiers

✓ Les opérateurs : +, -, *, /, %, etc avec les règles comme en C/C++

0.4 Types de base

• Tout objet doit être créé (sauf pour les types de base, pour faciliter les choses).

Type primitif	Taille	Min	Max	Type non basic (Wrapper)
boolean	–	–	–	Boolean
char	16 bits	Unicode 0	Unicode $2^{16}-1$	Character
byte	8 bits	-128	+127	Byte
short	16 bits	-2^{15}	$+2^{15}-1$	Short
int	32 bits	-2^{31}	$+2^{31}-1$	Integer
long	64 bits	-2^{63}	$+2^{63}-1$	Long
float	32 bits	IEEE754	IEEE754	Float
double	64 bits	IEEE754	IEEE754	Double
void	–	–	–	Void

• Exemple :

```
char c = 'x'; Character ch = new Character(c);
```

Ou

```
Character ch = new Character('x');
```

• Pour *String* (basic) :

```
String s = "asdf";
```

ou

```
String s = new String("asdf");
```

0.4.1 Valeur des type par défaut des types primitifs

• Déclarées dans les classes ou les fonctions :

Type	Valeur par Défaut
boolean	false
char	le caractère nul $\backslash u0000$ (null)
byte	(byote) 0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

0.4.2 A propos de new

Qu'est ce que l'allocation dynamique ?

Rappel : tout objet doit être créé (avec new)

Dans `String s = new String("asdf");`

- Si 's' est local, il disparait mais la zone reste reste occupée mais non accessible.

- Comme en C/C++,

- En C/C++, le programmeur doit s'en occuper

- En Java, on a le ramasse-miettes *garbage collector* = *gc*.

0.5 Variable, Attribut ou Méthode "Static"

- *static* : non attaché à un objet particulier instance d'une classe mais exemplaire unique.

  ➥ En C++, on l'appelle donnée ou variable de classe

- Une méthode static peut être appelée via un objet instance ou via le nom de la classe.

- Un des intérêts d'une méthode "static" est de ne pas créer d'objet pour pouvoir l'appeler.

  ➥ C'est toujours le cas de la fonction "main".

- Exemple : la fonction main est *static*

```
// Affichage de la date du jour
import java.util.*;
public class QuelleDate {
  public static void main(String[] args) {
    System.out.println("Salut ! on est : ");
    System.out.println(new Date());
  }
}
```

0.6 Opérateuts de comparaison, if(...)

```
import java.util.Scanner;
public class Comparaison
{
   public static void main( String args[] )
   {
      Scanner input = new Scanner( System.in );
      int nombre1;
      int nombre2;
      System.out.print( "Entrez  1er entier : " );
      nombre1 = input.nextInt();
      System.out.print( "Entrez 2e   entier: " );
      nombre2 = input.nextInt();
      if ( nombre1 == nombre2 )    System.out.printf( "%d == %d\n", nombre1, nombre2 );
      if ( nombre1 != nombre2 )    System.out.printf( "%d != %d\n", nombre1, nombre2 );
      if ( nombre1 < nombre2 )     System.out.printf( "%d < %d\n", nombre1, nombre2 );
      if ( nombre1 > nombre2 )     System.out.printf( "%d > %d\n", nombre1, nombre2 );
      if ( nombre1 <= nombre2 )    System.out.printf( "%d <= %d\n", nombre1, nombre2 );
      if ( nombre1 >= nombre2 )    System.out.printf( "%d >= %d\n", nombre1, nombre2 );
   }
}

Sortie : Entrez  1er entier : 32
      Entrez 2e   entier: 32
      32 == 32
      32 <= 32
      32 >= 32
```

Listing 5 – Comparaison de nombres

0.7 Style et Convention (suite)

✓ if (...) : la condition entre parenthèses

✓ Ne pas confondre '==' et '='

✓ Quand '=' participe dans une comparaison, il est à droite (==, <=, >=, !=)

✓ Table des priorités de quelques opérateurs :

Opérateur	Associativité	Type
* / %	Gauche à droite	multiplicatif
+ -	Gauche à droite	additif
< <= > >=	Gauche à droite	relational
== !=	Gauche à droite	égalité
=	Droite à Gauche	affectation

0.8 Exemples & Exercices

• 1- Ecrire une application qui lit un diamètre (entier) de cercle et affiche la superficie et le périmètre du cercle.

• 2- Ecrire une application qui lit un min et un max et qui affiche la table des carrés, cubes, des racines carrés et les log_e des nombres entiers entre min et max.

- Solution (pour les deux) sous forme de deux fonctions *cercle()* et *table()*.
- Utilisation du package *java.lang.Math*
- Boucle for, itérations, variable i

• Les deux fonctions sont static car on ne peut appeler une non-static depuis une static.

```
2
4
    import java.util.*;
6   import java.lang.Math;

8 public class Cercle_et_tableau {
        public static void main(String[] args) {
10      cercle();
        table();
12 }

14      public static void cercle() {
        java.util.Scanner in = new java.util.Scanner(System.in);
16      System.out.println("quel est le diametre ?");
        double diametre= in.nextInt();
18      System.out.printf("Diamètre= %f, Périmètre=%f, Aire=%f \n",
                diametre, diametre * java.lang.Math.PI, diametre*diametre/4 * java.lang.Math.PI);
20
}
22      public static void table() {
        java.util.Scanner in = new java.util.Scanner(System.in);
24      System.out.println("Donner le min et le max ? ");
        int min= in.nextInt();
26      int max= in.nextInt();

28      System.out.printf(" I  \t  I*I \t I*I*I*I \t     sqrt(I) \t    LOGe(I)\n");
        System.out.printf("-------------------------------------------------------------\n");
30
        for (int i=min; i<= max; i++)
32          System.out.printf(" %3d \t  %4d \t  %8.0f \t   %8.2f \t  %8.2f\n", i, i*i,
                java.lang.Math.pow((double)i,3.0), java.lang.Math.sqrt(i), java.lang.Math.log(i));
```

```
34 }
}
```

- **Exécution :**

```
quel est le diametre ?
4
Diamètre= 4,000000, Périmètre=12,566371, Aire=12,566371
Donner le min et le max ?
1 10
 I        I*I       I*I*I*I           sqrt(I)        LOGe(I)
-------------------------------------------------------------
   1          1           1             1,00          0,00
   2          4           8             1,41          0,69
   3          9          27             1,73          1,10
   4         16          64             2,00          1,39
   5         25         125             2,24          1,61
   6         36         216             2,45          1,79
   7         49         343             2,65          1,95
   8         64         512             2,83          2,08
   9         81         729             3,00          2,20
  10        100        1000             3,16          2,30
```

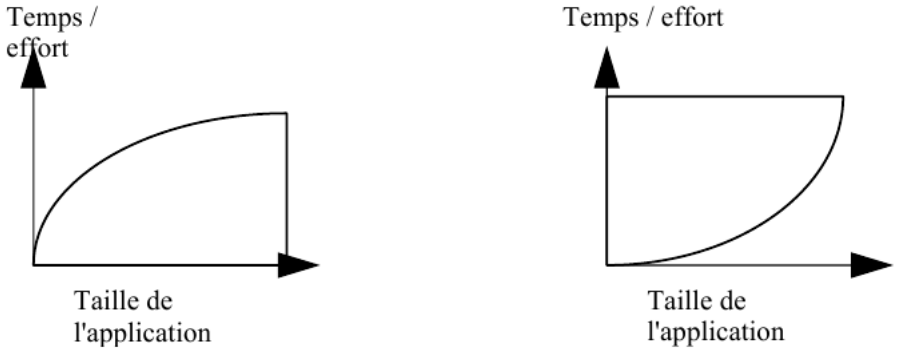
0.9 POO : Classes et Objets

0.9.1 Objectifs de la programmation orientée objets (POO)

✓ **Modélisation** directe des objets du monde réel par des entités informatiques

✓ Exploitation de la **redondance** :

dans le monde réel, on a de nombreux représentants de peu de concepts différents



Comparaison des coûts : approches Objets et Traditionnelle

✓ **Réutilisation** des composants logiciels, des composants préexistants comme en électronique, en construction automobile, etc.

✓ **Réduction du couplage** et dépendances entre différentes parties d'une application par une interface minimale et clairement définie entre objets

✓ **Protection** et **abstraction** des entités par la séparation de l'interface et de l'implantation

0.9.2 Les principes de la POO

✘ **Tout est objet** (contenant un état, une identité et un comportement = des compétences)

➥ Un objet "rend service" ! ➙ l'objet *Date* donne la date.

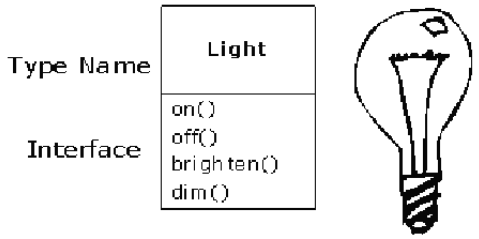
✘ Un programme est un ensemble d'objets communicants ("quoi faire") à travers des messages

✘ Chaque objet a sa propre mémoire, remplie d'autres objets

✘ Tout objet a un type

✘ Tous les objets d'un même type peuvent recevoir le même message.

✘ Chaque objet a une interface (voir le dessin) et une implantation (cachée)



Exemple en java :

```
Light lampe = new Light();
lampe.on();
```

0.9.3 Objets

– Un objet est un morceau de données

– Contrairement aux données passives qui sont manipulées par des procédures, un objet peut être une donnée active;

– Un objet peut être un nombre, un mot, une feuille de calcul, un tableur ou l'image d'un circuit électronique;

➥ Si c'est un nombre, il peut savoir se doubler, calculer son opposé ou se multiplier par pi;

– On effectue ces actions en envoyant un message à l'objet lui disant de faire telle action.

Par exemple : *dessin.afficher()* on demande au dessin de s'afficher

1 + 2 on envoie le message "+" à 1 avec le paramètre 2

– Les objets sont des entités privées, ils se manipulent et se modifient eux mêmes; leur interface est propre est claire;

Ils envoient et reçoivent des messages; rien d'autre.

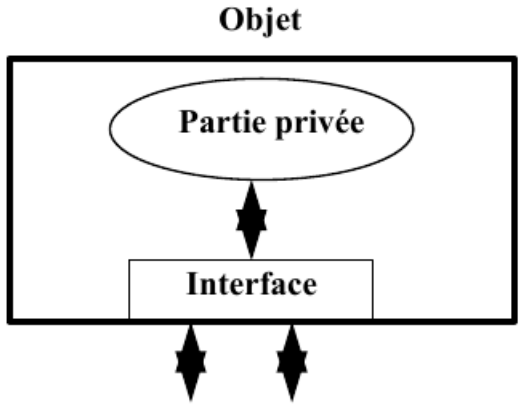
0.9.4 Encapsulation : une notion importante

Regrouper dans une même entité ce qui concerne un objet.

✓ Regroupement des données et traitements dans une entité logiquement homogène

✓ Mise en place par les objets, elle permet de développer des applications complexes par des objets indépendants et faciles à déverminer (déboguer).

➧ Si un objet **Roue** est créé, testé et validé, ça ne sera plus nécessaire de la réinventer à chaque fois que l'on a besoin d'une roue.



0.9.5 Classe : une ontologie

✓ Une classe est un groupe d'objets qui partagent le même comportement et propriétés.

Par exemple, dans la classe animal :

Si un animal peut respirer, manger et se reproduire, alors tous les animaux (de sa classe) peuvent en faire autant.

✓ On peut créer une sous classe d'objets qui possèdent des propriétés particulières.

Par exemple, les mammifères constituent une sous classe de la classe animal avec leur propre propriétés : **les mères produisent du lait pour les petits.**

✓ Les sous classes peuvent avoir des sous classes.

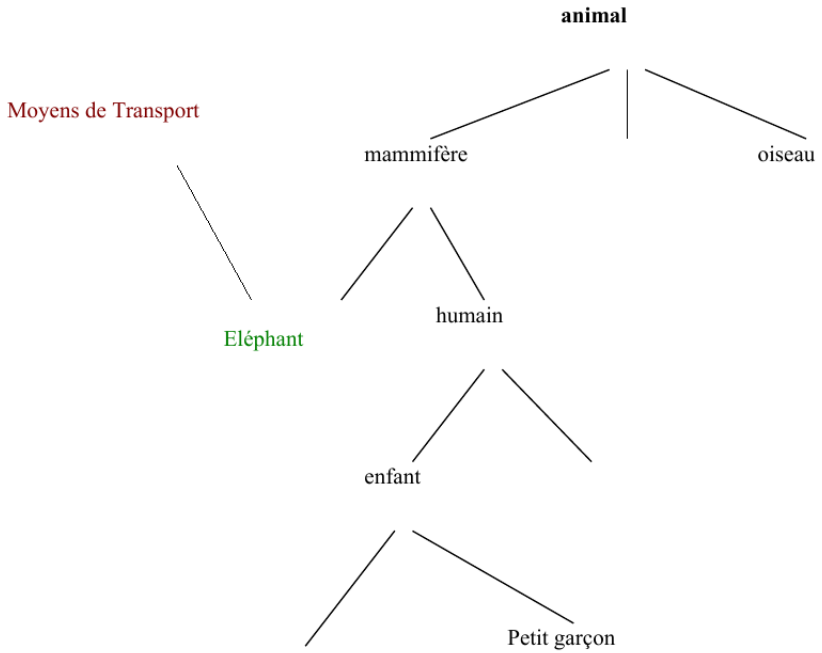
P. Ex., on peut passer des mammifères aux humains, aux enfants puis aux petits garçons, etc.

Chaque sous classe **hérite** des propriétés des ses classes parents (de ses super classes).

Graphe animal :

- Puisque les animaux peuvent manger, les petits garçons le peuvent aussi.
- Si l'on ajoute une propriété à une classe, cette nouvelle capacité est héritée par toutes ses sous classes.

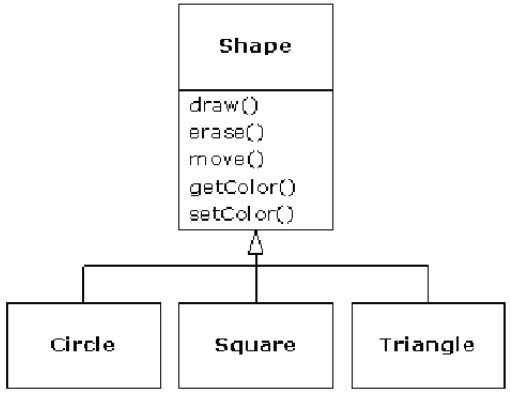
➥ Par exemple, si les animaux peuvent voler comme superman, alors les petits



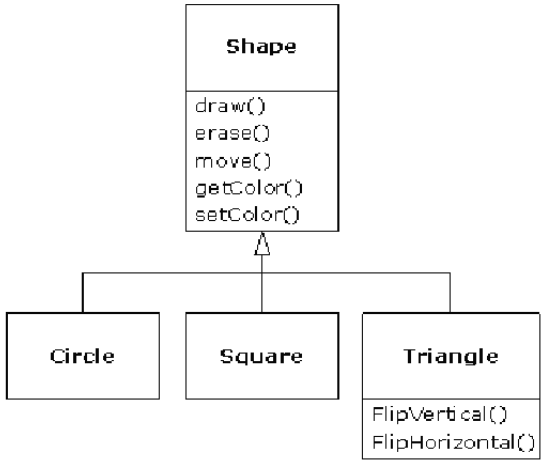
0.9.6 Un exemple d'héritage (UML)

Héritage : qu'est ? Ontologie ?

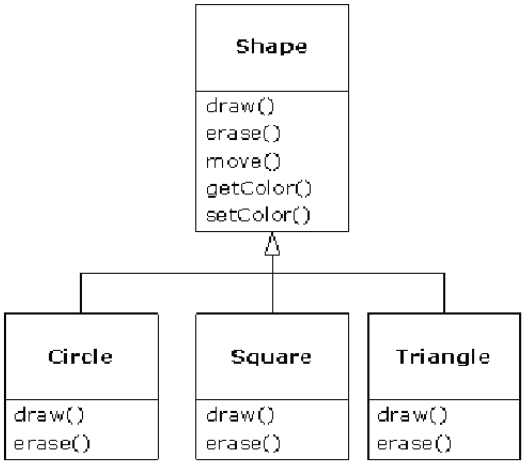
Exemple : classe *Forme* (géométrique)



Héritage



Spécialisation du Triangle



Surcharge des fonctions

- On peut ensuite spécialiser le Triangle (qui n'a pas de sens pour les autres)
- On ré écrit les (mêmes) méthodes pour chaque sous classe :

  ➥ tout le monde sait faire draw() mais chacun à sa manière !

0.9.7 Les objets de l'univers du problème et leurs relations

✓ Les objets ont des propriétés que nous percevons comme des vérités premières.

✓ Ces objets sont représentés de manière structurée faisant appel à des notions telles que :

Classification (est-un)

Rattache un objet à son type générique

Exemple : *marie est un étudiant (l'objet marie de la classe étudiant)*

Agrégation (partie-de)

Relie un objet à ses composants

Exemple : *pied de table*

Généralisation (sorte de)

Relie les catégories d'objets à d'autres plus génériques (héritage)

Exemple : *un étudiant est une personne adulte*

0.9.7.1 Exemple d'un graphe de relations

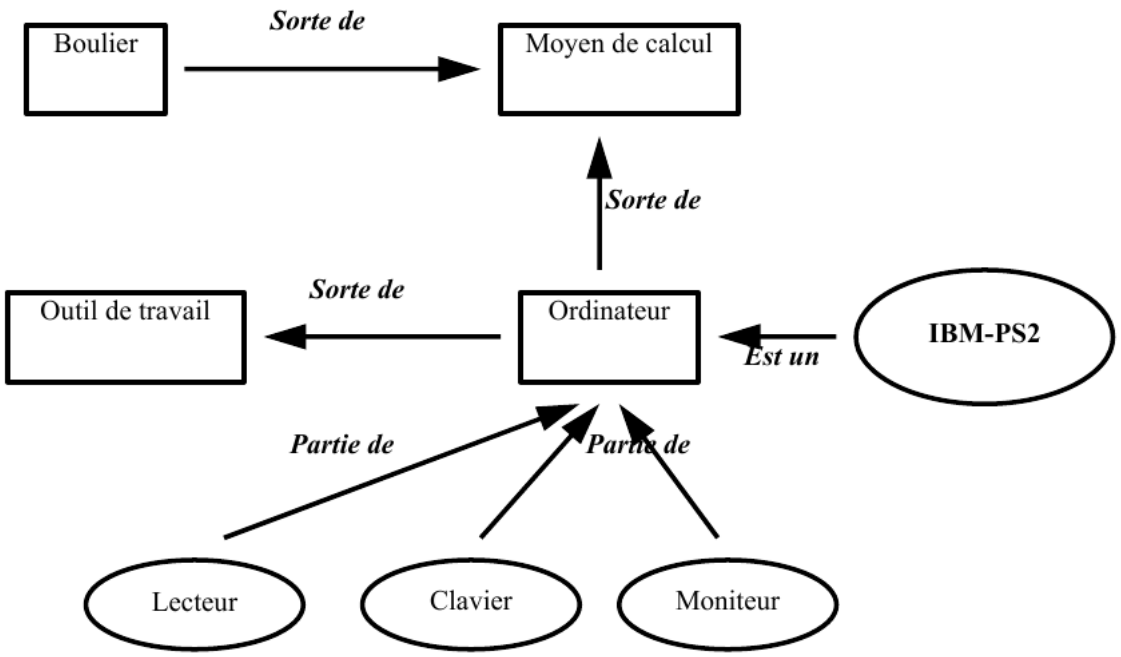


FIG. 1 – Un exemple de graphe de relations

Si l'on ajoute : "l'ordinateur est une sorte d'objet de décoration", alors l'objet IBM-PS2 sera aussi un objet de décoration !

0.9.8 Le monde des objets : vocabulaire

Monde = collection d'objets

Objet = attributs + méthodes

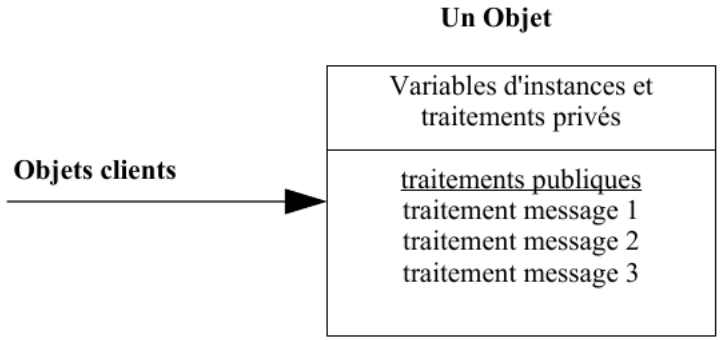
Attributs = variables d'état d'objet

Méthodes = traitements = compétences de l'objet

Message = sélecteur + arguments

Sélecteur = indication d'une méthode de l'objet

Objet client = expéditeur du message



0.9.9 Concepts de base de la POO

✓ **Encapsulation**

- ◗ Objet = données + traitements
- ◗ traitement = méthode

✓ **Protection des données et traitements**

- ◗ Objet = interface + implantation

✓ **Masquage de l'implantation**

- ◗ implantation fournie compilée

✓ **Classe et instances**

- ◗ classe = concept = type
- ◗ instance = représentant du concept = variable
- ◗ factorisation des traitement dans les classes

✓ **Communication par envoi de messages**

◗ un objet envoie un message à un autre pour lui demander d'activer une méthode

✓ **Héritage de classes**

◗ Réutilisation et extension de classes existantes

◗ Une sous classe d'une classe hérite des données et des traitements de la super classe.

✓ **Polymorphisme**

◗ Manipulation de données polymorphes

◗ Un objet polymorphe est un objet pouvant prendre différentes formes (différents types)

◗ Un traitement porté sur un objet polymorphe est un traitement polymorphe :

➥ le traitement peut s'appliquer invariablement sur différents types d'objets (exemple : addition des réels et des entiers)

0.10 Historique des langages objets

✓ Simula dans les années 60

✓ Smalltalk dans les années 70 (pur langage objet, définitions des 1ers interfaces graphiques)

✓ Langages objets natifs (purs) et langages à extension objet

➥ natif : le langage ne manipule que des objets (cf. Smalltalk)

➥ extension objet des langages existants (pratiquement tous les langages classiques) :

◗ C à Objective C, C++ (Java ?), Pascal à Pascal objet

◗ Ada à Ada 95

◗ Lisp à Falvors, Xlisp, common Lisp, clos

◗ Prolog à L&O, Emicat, ...

✓ Des langages tels que Basic, Fortran et Cobol (etc.) proposent aussi des objets.

Nous allons étudier le langage **Java** (par endroit en le comparant à C/C++)

0.11 Classe et objet en java

0.11.1 Une classe simple : annuaire cours (+ affichage)

```
1
3
public class Annuaire_cours
5 {
7    public void affiche()
     {
9       System.out.println( "Bienvenu dans l'annuaire de cours !" );
     }
11 }
```

- Puis (utilisation) :

```
1
3
public class Test_Annuaire_cours
5 {
7    public static void main( String args[] )
     {
9
        Annuaire_cours mon_annu = new Annuaire_cours();
11
        mon_annu.affiche();
```

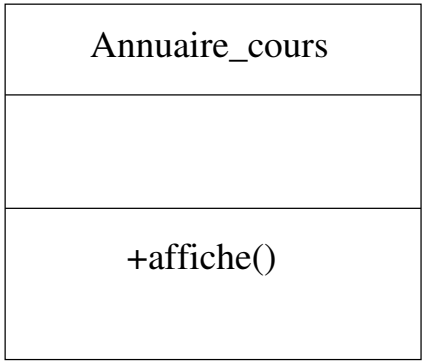
```
13      }
}
```

➥ Exécution : `Bienvenu dans l'annuaire de cours !`

• Règle : on ne peut déclarer plus d'une classe publique dans un fichier java.

➥ Le nom du fichier = le nom de la classe (+ *.java*)

UML : représentation d'une classe.



0.11.2 *Affichage d'un paramètre de fonction avec printf*

```
1
3 public class Annuaire_cours1
  {
5
      public void affiche(String cours)
7     {
          System.out.printf( "Bienvenu a l'annuaire du cours  %s \n", cours + " !!!");
9     }
  }
```

- Puis (utilisation) :

```
2
4
  import java.util.Scanner;
6 public class Test_Annuaire_cours1 {

8     public static void main( String args[] )
      {
10
          Scanner input = new Scanner( System.in );
12
          Annuaire_cours1 mon_annu = new Annuaire_cours1();
14
          System.out.println( "Entrez un nom de cours :" );
16        String nomDeCours = input.nextLine();
```

```
        System.out.println();
18
        mon_annu.affiche( nomDeCours );
20    }
}
```

- Exécution :

```
Entrez un nom de cours :
  info
Bienvenu à l'annuaire du cours  info!!!
```

0.11.3 Style et conventions (suite)

• On a dû importer *Scanner* par `import java.util.Scanner;`

• En règle générale, tout ce que l'on utilise doit être importé.

• Pourquoi pas *System* , *String* ou *Annuaire_cours* ?

➥ Les classes *System* et *String* sont dans le package **java.lang** , implicitement importé pour tout programme Java; *Annuaire_cours* importé par défaut car :

➥ Il y a une relation spéciale entre les classes compilées dans un même répertoire.

➧ Par défaut, ces classes sont considérées du même package.

• Pour *Scanner*, pas besoin d'import si l'on écrit **java.util.Scanner** dans toute utilisation.

➥ Par exemple (*qualification complète*) :

```
java.util.Scanner input = new java.util.Scanner( System.in );
```

0.11.4 *Variable d'instance et get et set*

```
2  public class Annuaire_cours2
   {    private String nomCours;
4
       public void setNomCours( String nom )
6      {
           nomCours = nom;
8      }

10     public String getNomCours()
       {
12         return nomCours;
       }
14
       public void affiche()
16     {
           System.out.printf( "Bienvenu à l'annuaire pour  %s!\n",  getNomCours() );
18     }
   }
```

• La partie (en anglais) *"set"* et *"get"* du nom des méthodes est une **convention** (voire, exigée par certaines outils comme JavaBeans).

◗ Ces méthodes sont ici publiques ➙ accessibles pour toute instance de *Annuaire_cours2*

◗ Elles peuvent être non public (*private, protected*) dans d'autres cas (cf. un compte bancaire) !

0.11.5 Style et conventions (suite)

- Faire précéder chaque *attribut* et *méthode* par un label d'accès.
- Une règle de base : les attributs en **private**, les méthodes en **public** .
  - ➥ Méthode *private* si leur accès doit être limité.
- Une donnée private n'est accessible que par les méthodes de sa classe.

```
XXX class colinMayar {
     private  void      methode_Cachee_A_L_Exterieur(...) {...}
     public  double    on_Peut_Appeller_Celle_Ci(...) {...}
     public   int      je_Suis_visible;
     private  String   je_Suis_Secret;
  }
```

- Dans l'exemple *Annuaire_cours2* : on a défini l'attribut avant les méthodes.
  - ➥ D'autres pourraient inverser cet ordre (question d'habitude/style)
  - ➥ Dans une interface, il n'y a pas d'ordre (même pas de précédence).

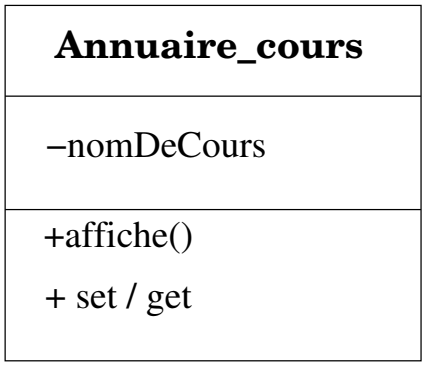
- Utilisation d'*Annuaire_cours2* :

```
1
3
import java.util.Scanner;
5 public class Test_Annuaire_cours2
{
7
    public static void main( String args[] )
9   {

11      Scanner input = new Scanner( System.in );

13      Annuaire_cours2 mon_annu  = new Annuaire_cours2();

15      System.out.printf( "La valeur actuelle du nom du cours est : %s\n\n",
            mon_annu.getNomCours() );
17
        System.out.println( "Entrez un nom de cours :" );
19      String nomDeCours = input.nextLine();

21        mon_annu.setNomCours(nomDeCours);

23      System.out.println();
        mon_annu.affiche();
25  }
}
```

◗ Exécution :

```
La valeur actuelle du nom du cours est :   null
Entrez un nom de cours :
  Java
Bienvenu à l'annuaire pour  Java!
```

UML : représentation augmentée d'une classe.



0.11.6 Type de référence (non primitif)

- Hormis les types primitifs, les (autres) types sont considérés comme des types de référence.

  ➥ Il faut donc utiliser *new* pour en créer une instance.

  ➥ Exemple : `Compte monCompte = new Compte(...);`

- Une variable du type *Compte* s'initialise par un **constructeur**

.

  ➥ C'est une fonction dont le nom = le nom de la classe, avec ou sans paramètres.

0.11.7 Constructeur et constructeur par défaut

Dans cet exemple, l'Annuaire_cours3 dispose de deux constructeurs.

```
2
4  public class Annuaire_cours3
   {
6      private String nomCours;

8      public Annuaire_cours3( String nom )
       {
10         nomCours = nom;
       }
12
       public Annuaire_cours3()
14     {
           nomCours = "par défaut";
16     }

18     public void setNomCours( String nom )
       {
20         nomCours = nom;
       }
22
       public String getNomCours()
24     {
           return nomCours;
26     }

28     public void affiche()
```

```
    {
30      System.out.printf( "Bienvenu à l'annuaire pour  %s!\n",  getNomCours() );
    }
32  }
```

Utilisation d'Annuaire3 :

```
2
import java.util.Scanner;
4 public class Test_Annuaire_cours3
  {
6
    public static void main( String args[] )
8   {
      Annuaire_cours3 mon_annu1  = new Annuaire_cours3("Informatique");
10    Annuaire_cours3 mon_annu2  = new Annuaire_cours3("Mathématiques");
      System.out.printf( "La valeur actuelle du nom du cours1 est : %s\n\n",
12        mon_annu1.getNomCours() );
      System.out.printf( "La valeur actuelle du nom du cours2 est : %s\n\n",
14        mon_annu2.getNomCours() );
      Annuaire_cours3 mon_annu3 =  new Annuaire_cours3();
16    System.out.printf( "La valeur actuelle du nom du cours3 est : %s\n\n",
          mon_annu3.getNomCours() );
18

20    System.out.println( "Entrez un nom du 3e cours :" );
      Scanner input = new Scanner( System.in );
22    String nomDeCours = input.nextLine();

24      mon_annu3.setNomCours(nomDeCours);
```

```
26          System.out.println();
            mon_annu3.affiche();
28      }
}
```

◗ Exécution :

```
La valeur actuelle du nom du cours1 est : Informatique
La valeur actuelle du nom du cours2 est : Mathématiques
La valeur actuelle du nom du cours3 est : par défaut
Entrez un nom du 3e cours :
   Sport
Bienvenu à l'annuaire pour  Sport!
```

0.11.8 Un autre exemple : classe Compte

- Type double précision **double** pour plus de précsion (en particulier dans les opérations)

```
2
public class Compte
4 {
    private double solde;
6
    public Compte( double SoldeInit )
8   {

10      if ( SoldeInit > 0.0 )   solde = SoldeInit;
    }
12
    public void credit( double montant )
14  {       solde = solde + montant;
    }
16
    public double getSolde()
18  {
        return solde;
20  }
}
```

➥ Initialisation à un solde non négatif (pour rester initialisé à 0.0 par défaut).

- Utilisation de Compte bancaire :

```
1
3  import java.util.Scanner;
   public class Test_Compte
5  {
7     public static void main( String args[] )
      {
9        Compte compte1 = new Compte( 50.00 );
         Compte compte2 = new Compte( -7.53 );
11
         System.out.printf( "Solde compte1 :$%.2f\n", compte1.getSolde() );
13       System.out.printf( "Solde compte2 :$%.2f\n\n",compte2.getSolde() );

15       Scanner input = new Scanner( System.in );

17         double montantDepot;
           System.out.print( "Enterer montant depot pour compte1: " );
19         montantDepot = input.nextDouble();
           System.out.printf( "\najout de %.2f à compte1 \n\n",montantDepot );
21         compte1.credit( montantDepot );

23         System.out.printf( "Solde compte1 :$%.2f\n",compte1.getSolde() );
           System.out.printf( "Solde compte2 :$%.2f\n\n",  compte2.getSolde() );
25         System.out.print( "Enterer  montant depot pour compte2: " );
           montantDepot = input.nextDouble();
27         System.out.printf(  "\najout de %.2f à compte2 \n\n",montantDepot );
           compte2.credit( montantDepot );
29
           System.out.printf( "Solde compte1 :$%.2f\n", compte1.getSolde() );
31         System.out.printf( "Solde compte2 :$%.2f\n",   compte2.getSolde() );
      }
33 }
```

◗ **Exécution** :

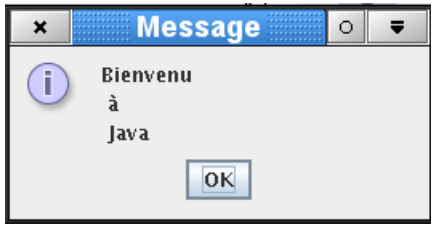
```
Solde compte1 :$50,00
Solde compte2 :$0,00
Enterer montant depot pour compte1: 30
ajout de 30,00 à compte1
Solde compte1 :$80,00
Solde compte2 :$0,00
Enterer  montant depot pour compte2: 32
ajout de 32,00 à compte2
Solde compte1 :$80,00
Solde compte2 :$32,00
```

0.12 Graphisme simple avec Java

• Affichage d'une fenêtre de dialogue simple :

```
2
import javax.swing.JOptionPane;
4 public class Dialogue1
{
6    public static void main( String args[] )
     {
8
        JOptionPane.showMessageDialog( null, "Bienvenu \n à \n Java" );
10   }
}
```

➥ La sortie (On clique pour sortir !)

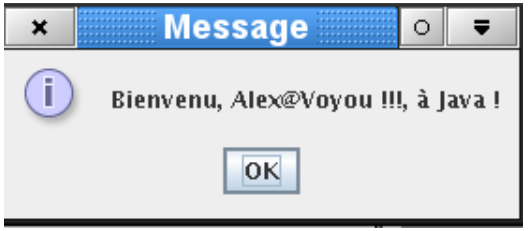


• Pas de création d'ojet de la classe *JOptionPane* car la méthode *showMessageDialog* de cette classe est **static**.

0.12.1 Graphisme avec saisi

```
1
3 import javax.swing.JOptionPane;
  public class Dialogue2
5 {
      public static void main( String args[] )
7     {
9         String nom = JOptionPane.showInputDialog( "Quel est ton nom, mon petit?" );
11        String message =  String.format( "Bienvenu, %s, à Java !", nom );
13        JOptionPane.showMessageDialog( null, message );
      }
15 }
```

➥ La sortie



➥ La variable *nom* récupère la réponse qui sera affichée.

0.12.2 *Un autre exemple de Graphisme avec saisi et calculs*

```
1
3 import javax.swing.*;
5 public class ex_lect_graphique
  {
7
     public static void main( String args[] )
9    {
11      String s = JOptionPane.showInputDialog(null,"Entrer un diamèter: ");
        double x = Double.parseDouble(s);
13      JOptionPane.showMessageDialog(null,"Le périmètre =  " + 3.14*x);
      }
15 }
```

- Même principe que l'exemple précédent mais avec un calcul de périmètre.

0.12.3 Exercice et solution

• Modifier l'exercice sur l'addition et ajouter des boites de dialogue.

NB. : la méthode showInputDialog renvie un String. Il faut donc convertir cette valeur en un entier (int) pour les calculs.

➥ La métode **int Integer.parseInt(String s)** peut servir.

➥ Si le String ne contient pas un entier valide, il y aura une erreur + fin de programme.

```
1

3 import javax.swing.JOptionPane;
  public class Dialogue3
5 {
    public static void main( String args[] )
7   {

9       String chVal = JOptionPane.showInputDialog( "Donne un entier, mon petit?" );

11

        String message =  String.format( "Son successeur est  %s !", Integer.parseInt(chVal)+1);
13
        JOptionPane.showMessageDialog( null, message );
15
```

```
17

    }
19 }
```

N.B. : les deux dernières lignes peuvent être remplacées par :

```
JOptionPane.showMessageDialog( null, "Son successeur est " +
(Integer.parseInt(chVal)+1) + " !!");
```

➥ Il faut parenthéser *(Integer.parseInt(chVal)+1)*; sinon, le "+" sera pris pour une concaténation !

0.13 Un 1er bilan

• Classe, attribut, méthode, instance, ...

• Appel d'une Méthode = envoi de message, ...

• Toute classe *public* doit être placée dans un fichier Java du même nom

• Le mot clef *public* est un modifieur d'accès

• Une classe / méthode *public* est accéssible au public :

➥ Elle peut être utilisée / appelée par quiconque.

• Convention de nommage : *ceciEstUnNomDeClasseTresLong*

• Toute classe contenant `public static void main( String args[] )` peut être appelée comme une application (main = point d'entrée)

• On ne peut appeler les méthodes "public" d'une classe que si l'on crée un objet (avec new) instance de cette classe (sauf pour une classe "static").

0.14 La classe Scanner (entrée sortie)

• La méthode *nextLine()* de *Scanner* lit une ligne et renvoie un *String*

• La méthode *next()* lit des caractères jusqu'à un espace et renvoie les caractères lus sous forme d'un *String*

• La méthode *nextDouble()* lit le prochain réel double précision

• La méthode *nextInt()* lit le prochain entier

➥ Plus généralement, la méthode *nextXxx()* lit le prochain xxx (de type primitif).

• Pour les mêmes méthodes, on a le test *hasNextXxx*...

0.15 La clause import

• Import est inutile si l'on qualifie entièrement un identificateur de classe

• La classe String est importée implicitement (par le *package java.langue*).

• Les classes compilées dans un même répertoire sont considérées comme faisant partie d'un même package par défaut.

➥ Ces classes sont importées implicitement dans toute autre fichier java définissant d'autres classes dans le même répertoire.

➙ Il n'est donc pas nécessaire de les importer.

0.16 classes et attributs

• Chaque instance d'une classe a ses propres copies des attributs.

• La plupart des attributs d'une classe sont *private* (attributs cachés / protégés).

➥ Ces attributs ne sont accessible que par les méthodes de la classe.

➥ Habituellement, les classes fournissent des méthodes publiques d'accès (get/set) aux attributs privés.

• Une différence entre les attributs et les variables locales / paramètres :

➥ les attributs sont initialisés par Java, pas les variables locales !

➥ Exemple : `String S` comme attribut (init null) / ou comme variable locale (non init !).

0.17 Types (suite)

• Les types de données sont divisés en 2 catégories : **primitifs** et **référence**.

➥ Les attributs de types primitifs *numériques* (byte , char , short , int , long , float, double) sont par défaut initialisés à *0*, le *boolean* à *false*.

• Une variable d'un type primitif ne peut contenir qu'une valeur à la fois (pas de méthode !).

➥ Les types de référence correspondent aux classes.

➥ Leur création passe par new et ils sont manipulés par leur adresse (init *null*) en mémoire.

• Un objet instance peut contenir de multiples attributs et méthodes.

• Un *constructeur* peut servir à initialiser une instance (d'une classe) lors de sa création.

➥ Un constructeur par défaut est fourni par Java (si l'on n'en fournit aucun).

• Un constructeur peut avoir des paramètres (comme tout autre méthode), mais pas de valeur de retour.

0.17.1 Promotion de types

- Promotion de type (conversion implicte) : **double > float > long > int > char.**

  ➥ **double > float > long > int > short > byte** (un short/byte n'est pas converti en char).

  ➥ Le type **boolean** de Java n'est pas convertible.

- Les conversions explicites comme en C.

0.18 A propos de printf

- Dans la méthode `Scanner.printf` :
  - %f représente les réels (float et double)
  - %s représente les strings (chaines de caractères), avec '+' éventuellement.
  - %b représente les booléens
  - %c les caractères, %d les entiers, ...

0.19 A propos du type String

• L'opérateur '+' permet de concaténer des strings (fait appel à la méthode **toString** définies dans toutes les classes fournies).

➥*String S="Resultat = " + " " + unEntier + " " + unReel + " "+ ..+ " "+ unBool_trueOrFalse;*

Donnera un string contenant la forme imprimable des ces variables.

• Attention : `int y=5; String Ch="y + 2 = " + y + 2;` donnera la chaîne "y + 2 = 52";

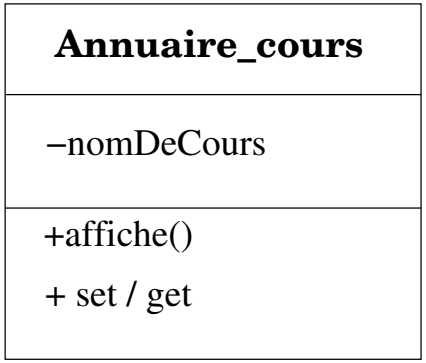
➥ Mais `String Ch="y + 2 = " + (y + 2);` donnera la chaîne "y + 2 = 7";

• Quelques exemples de String:

```
String string1 = new String("littéral valide");
String string2 = "littéral \n On a va à la ligne valide";
String string3 = "Joindre un  str" + "ing";
String string4 = "\"Escape Sequences\"\r"; // échappement
```

0.20 UML (suite)

- En UML : 3 champs pour une classe, le signe '+' pour public et '-' pour private...



0.21 fonction main : un exemple Java avec deux points d'entrées

```
1
3
import java.util.*;
5 public class test_double_main {
    public static void main(String[] args) {
7     System.out.println("Une première Date : ");
      System.out.println(new Date());
9      test_double_main2.main(null);

11    test_double_main2 test=new test_double_main2();
      test.main(null);
13  }
}
15
class test_double_main2 {
17  public static void main(String[] args) {
      System.out.println("Seconde Date Date : ");
19    System.out.println(new Date());
    }
21 }
```

➥ On remarque les deux manières d'appeler main de `test_double_main2`

• Exécution :

```
> java test_double_main
                        Une première Date :
                      Tue Dec 16 16:57:13 CET 2008
                        Seconde Date Date :
                        Tue Dec 16 16:57:13 CET 2008
                        Seconde Date Date :
                        Tue Dec 16 16:57:13 CET 2008


> java test_double_main2
                        Seconde Date Date :
                        Tue Dec 16 16:57:34 CET 2008
```

• N'importe quelle classes (public ou amicale) peut définir une méthode **main**.

0.22 Structures de Controle Java

- if (...) { ...} entre { } = un **bloc**.
- if (...) { ...} else {...} Attention à ';' dans `if (condition); ...`

Exemple :

```
if ( note >= 10 )
        System.out.println( "Passe !" );
else
        System.out.println( "Passe Pas !" );
```

- if (...) { ...}
  else if (...) {...}
  ...
  else {...}

N.B. : else fait référence à un if précédent : `if (A) if (B) {....} else {....};`

➥ Utiliser { } pour modifier cet effet : `if (A) {if (B) {....}} else {....};`

- Opérateur ? : comme raccourcis de if/else

```
System.out.println( note >= 10 ? "Passe" : "Passe Pas !" );
```

Autre Ex. : pour calculer le maximum entre deux entiers X et Y dans Max :

```
Max = X > Y ? X : Y;
```

- Schéma **while**, **for**, **do {...} while(..)**, **switch / case / break / default**, **break** et **continue**, ...

  ➥ N.B. : opérateurs conditionnels logiques : **&&**, **||** avec court-circuit

  ➥ Pour forcer l'évaluation des deux opérandes, on utilise les opérateurs logiques : **&** et **|**

    ➤ but : effet de bord : `if (sexe == Fem ) & ( ++x > 1) ...`

- L'opérateur ^ est un Ou exclusif. ! est la négation (comme en C)

- Le symbole de format %b dans *printf* permet d'écrire des booléens :

  ➥ `printf("%b ...", (jour == Lundi), ...);`

0.22.1 Exercices

- Exercice : Afficher les tables de vérités des opérateurs logiques ci-dessus.
- Exercice : lire N notes et calculer leur moyenne.

  ➥ Attention : l'opérateur '/' a le même comportement qu'en C/C++.

  ➥ Notion de conversion implicite vers le type plus fort (comme en C)

Remarques utiles aux exercices :

- N.B. : lecteur jusqu'à fin fichier (sur un Scanner) par

  while (input.hasNext()) { x = nextInt();}

- Opérateurs composés : +=, -=, *=, /=, %= (comme en C)
- Opérateurs ++ et – (en pré / post) comme en C

0.22.2 Exercices (suit)

• Ecrire un programme java qui lit n et calcule $n!$. puis

• Ecrire un programme Java qui calcule le valeur de e par :

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + ... + \frac{1}{n!}$$

• Ecrire un programme Java qui calcule le valeur de e^x par :

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + ... + \frac{x^n}{n!}$$

• Ecrire un programme Java qui calcule le capital restant R (initial + intérêts) d'un capital d'investissement initial C pour n années avec un taux annuel de t par $R = C.(1 + t)^n$

➥ On doit produire R_i pour chaque année $i \in 1..n$

➥ En java, **Math.pow(X, Y)** permet de calcule X^Y

0.22.3 Exercices : racine carée

Calculer la valeur approchée de la racine carée de l'entier A (A entre 1 et 100) par la méthode suivante :

- $U_0 = \frac{A}{2}$
- $U_n = \frac{1}{2} * (U_{n-1} + \frac{A}{U_{n-1}})$ pour $n > 0$

➥ On arrête les itérations lorsque $|U_n^2 - A| < \varepsilon$

• Le nombre trouvé est une valeur approchée de $\sqrt{A}$ car

$$|U_n^2 - A| < \varepsilon \text{ implique que } |U_n - \sqrt{A}| < \frac{\varepsilon}{U_n + \sqrt{A}} \quad \text{pour } U_n \approx \sqrt{A}$$

• On peut prendre $\varepsilon = 10^{-6}$

0.23 Exemple graphique : repère

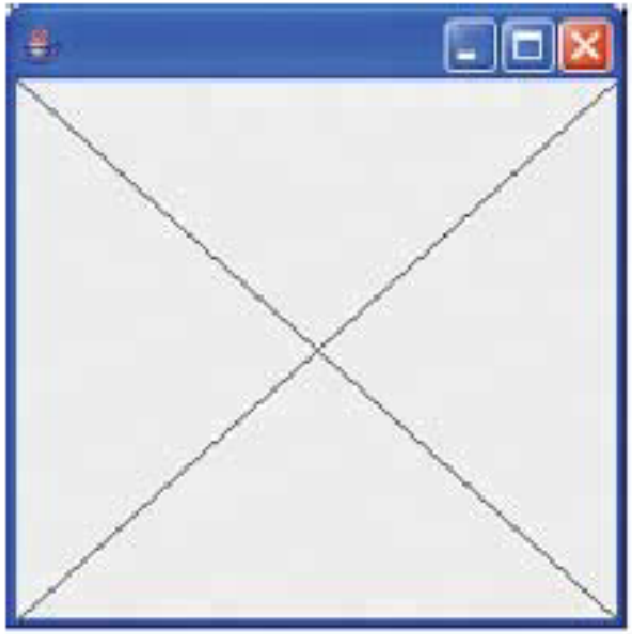
```
2 import java.awt.Graphics;
  import javax.swing.JPanel;
4 public class Repere extends JPanel
  {
6
     public void paintComponent( Graphics g )
8    {

10      super.paintComponent( g );
        int width = getWidth();
12      int height = getHeight();

14      g.drawLine( 0, 0, width, height );

16      g.drawLine( 0, height, width, 0 );
     }
18 }
```

- Utilisation

```
2
  import javax.swing.JFrame;
4 public class Test_repere
  {
6    public static void main( String args[] )
     {
8
        Repere panel = new Repere();
```

```
10
        JFrame application = new JFrame();
12
        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
14      application.add( panel );
        application.setSize( 250, 250 );
16      application.setVisible( true );
    }
18 }
```

➥ Sortie : une fenetre avec une croix



0.24 Tableaux Java

- Package : **java.util.Collections**
- Tableaux Statiques (taille non modifiable) :

➙ Déclaration et initialisation de tableaux

```
// Déclaration et initialisation de tableaux
int    tab1[] = {5, 10, 15, 20, 25};
String tab2[] = {"lundi", "mardi", "mercredi", "jeudi", "vendredi"};
```

Ne pas écrire (comme en C/C++) : `int tab[ 12 ];` ➙ erreur de compilation.

➙ Déclaration de tableaux sans initialisation

```
int tab3[];

// Création et Initialisation du tableau d'entiers
tab3 = new int[4];
tab3[0] = 28;
tab3[1] = 17;
tab3[2] = 150;
tab3[3] = 19;
```

➙ Tableaux de String

```
// Initialisation d'un tableau de chaînes
String tab4[];
tab4 = new String[3];
tab4[0] = "janvier";
tab4[1] = "février";
tab4[2] = "mars";
```

0.24.1 Vector : tableau dynamique

Création par **new**.

```
      Vector vect = new Vector;          // Vecteur d'Object
// OU
      Vector vect=new Vector(n);          // N cases !

// Ajout d'un élément :
          vect.addElement(o);

// Place d'un élément
         o=vect.elementAt(n);
```

0.24.2 Autres méthodes pour Vector

– *contains(Object)* : indique si l'objet est contenu dans le Vector.

– *copyInto(Object[])* : copie les éléments dans un tableau classique.

– *firstElement()* : renvoie le premier élément.

– *indexOf(Object)* : renvoie l'indice de l'objet

– *insertElementAt(Object, int)* : insère l'objet à l'indice indiqué

– *isEmpty()* : indique si le Vector est vide

– *lastElement()* : renvoie le dernier élément

– *removeAllElements()* : vide le Vector

– *removeElementAt(int)* : retire l'objet dont l'indice est donné

– *setElementAt(Object, int)* : place l'objet à l'indice donné

– *size()* : renvoie le nombre d'éléments

– ...

0.24.3 Exemple : les nombres Premiers

• Ecrire une application qui lit une valeur max et qui affiche les nombres premiers entre 1 et max.

Définition : P est premier s'il n'est divisible que par lui même et par 1.

- ◗ Encore : P est premier si aucun nombre inférieur à P ne le divise.
- ◗ Encore : P est premier si aucun nombre premier inférieur à P ne le divise.
- ◗ On peut démontrer que ces tests de division peuvent s'arrêter à $\sqrt{P}$.

➥ Indication : utiliser un tableau d'entiers pour retenir les nombres premiers précédents.

• Variante : lire min et max et donner les nombres premiers entre les deux.

➥ Variante avec tableau / sans tableau

N.B. : Appel de fonction avec paramètres

Vector et fonctions prédéfines sur *Vector*. (`vect.toString()` pour affichage)

Importance de faire *new Integer(..)*

```
1
3
    import java.util.*;
5   import java.lang.Math;

7 public class Prime {
    static Vector vect;
9
    public static void main(String[] args) {
11      java.util.Scanner in = new java.util.Scanner(System.in);
        System.out.println("quel est le max > 2 ?");
13      int valeur= in.nextInt();
        System.out.println("Les nombre premiers sont (1e Méthode) : ");
15      lesPrimes(valeur);System.out.println();

17      System.out.println("Les nombre premiers sont (2e Méthode) : ");
        lesPrimesAvecVect(valeur);System.out.println();
19
    }
21  public  static void lesPrimes(int val) {
        System.out.printf("(1), (2) ");
23       for (int i=3; i<= val; i+=2)
            {
25           if (! estMultiple(i, (int)(java.lang.Math.sqrt(i))))
                    System.out.printf(",(%d) ",i);
27          }
    }
29  public static boolean estMultiple(int val, int limite) {
            for (int i=2; i<= limite; i=i+1)
31              if ( val % i == 0) return true;
        return false;
33  }
```

```
35      public  static void lesPrimesAvecVect(int val) {

37           vect=new Vector();
             vect.add(1);vect.add(2);
39           for (int i=3; i<= val; i+=2)
                {
41               if (! possedUnMultipleDansVect(i, (int)(java.lang.Math.sqrt(i))))
                       {vect.addElement(i);
43                      }
                }
45          System.out.printf("\n Les résultats \n ");

47

            System.out.println(vect.toString());
49      }

51      public static boolean possedUnMultipleDansVect(int val, int limite) {
              for (int i=2; i < vect.size(); i++)
53                {if (val % (Integer)vect.elementAt(i) ==0 ) return true;
                   if ((Integer)vect.get(i) > limite) return false;
55                }
            return false;
57          }

59      }
```

- **Exécution :**

```
quel est le max > 2 ?
    25
Les nombre premiers sont (1e Méthode) :
     (1), (2) ,(3) ,(5) ,(7) ,(11) ,(13) ,(17) ,(19) ,(23)

Les nombre premiers sont (2e Méthode) :
Les résultats
    (1), (2), (3), (5), (7), (11), (13), (17), (19), (23),
```

0.25 Méthodes "static"

• La plupart des méthodes s'exécutent en réponse à un appel sur un objet spécifique.

• Parfois, une méthode réalise une action qui ne dépend pas d'un objet particulier.

➥ Une telle méthode s'applique d'une manière globale à l'objet qui la déclare : c'est une méthode **static**.

➥ Une classe peut avoir un groupe de méthodes static qui réalisent des tâches communes.

◗ Par exemple, la méthode **pow** de la classe Math

➥ On n'est pas obligé d'instancier la classe Math pour utiliser **pow**.

➥ Un autre exemple de la même classe est **sqrt** : *Math.sqrt(16.0)*

• N.B. : toutes les méthodes de la classe Math sont static.

• Rappel : chaque objet d'une classe *Cl* a ses propres copies des attributs de *Cl* sauf pour les attributs "static" de *Cl*

➥ Ces attributs static sont des exemplaire unique de données partagés par toutes les instances

➥ Ce sont appelés *attributs de classe* (eN C++, on dit *variables de classe*).

• Rappel : une méthode dans une classe peut appeler une autre méthode de la classe en évoquant son nom.

➥ Cependant, en JAVA, une méthode static d'une classe ne peut directement appeler qu'une autre méthode static de la même classe (sinon, erreur de compilation).

• De même, une méthode static ne peut manipuler directement que des attributs *static* de sa classe (car les attributs *static* sont commun à tous les objets instances de la classe).

➥ La raison : chaque instance a ses propres copies des attributs;

Si une méthode static appelle une méthode non-static, comment pourrait la méthode static savoir quelle instance manipuler ? Et si aucune instance n'existe encore lorsque l'on invoque la méthode non-static ?

➥ Pour ces raisons logiques, Java impose à une méthode static d'appeler et manipuler que des entités "static".

➥ Pour manipuler directemnt les attributs / méthodes non static, on est obligé de passer par une instance de la classe.

• La fonction **main** est static : on peut l'appeler sans devoir créer une instance.

➥ Au lancement, JVM charge toute la classe et utilise son nom pour invoquer la méthode main.

➥ Cet appel peut être suivi d'arguments séparés par un espace.

➥ La méthode main peut être placée dans n'importe quelle classe (pas forcément public). C'est le nom passé à JVM qui permet de décider.

➥ Cela permet de tester individuellement chaque classe.

0.25.1 Quelques une des méthodes de la classe Math

abs(23.4), abs(-32.2), ... : valeur absolue

ceil(double) / **floor**(double) : arrondi à l'entier supérieur / inférieur

cos(double), **sin**(double), **tan**(double), ...

exp(double) : e^x

pow(double, double) : lever à la puissance

log(double) : log_e ➙ log(*Math.E*)=1.0, log(Math.E * Math.E)=2.0

max(double, double)/**min** : le maximum de deux réels

• N.B. : **Math.PI** = 3.14..., **Math.E** = 2.718...

➥ Ces constantes sont déclarée **public final static**

➧ *public* pour être accessible, *final* = constante non modifiable, *static* pour être accessible par les fonctions *xx* : *Math.xx*

0.25.2 Exemple : maximum de 3 valeurs

- Ecrire un programme Java qui lit 3 réesl et trouve le maximum.

```
2  import java.util.Scanner;
   public class max_des_3
4  {
      public static void main(String args[] )
6     {
         Scanner input = new Scanner( System.in );
8
         System.out.print(
10          "Entrer 3 réels separés par spaces: " );
         double nombre1 = input.nextDouble();
12       double nombre2 = input.nextDouble();
         double nombre3 = input.nextDouble();
14
         double result = maximum( nombre1, nombre2, nombre3 );
16
         System.out.println( "Le Maximum =: " + result );
18       System.out.println( "Le Maximum par Math : " +
                     Math.max( nombre1, Math.max( nombre2, nombre3 ) ) );
20    }

22
      public static double maximum( double x, double y, double z )
24    {
         double valeurMaximum = x;
26
         if ( y > valeurMaximum ) valeurMaximum = y;
28
         if ( z > valeurMaximum ) valeurMaximum = z;
30       return valeurMaximum;
      }
32 }
```

- **Exécution :**

```
Entrer 3 réels separés par spaces: 12.5 2.7 16.9
Le Maximum =: 16.9
Le Maximum par Math : 16.9
```

Remarques :

-

  ➥ La méthode *main* doit être static

  ➥ Si *main* appelle une méthode (sans créer d'instance), la méthode appelée doit être static.

- N.B. : on aurait pu calculer le maximum de deux valeurs ditectement par *Math.maximum*

0.26 APIs Java

• Utilisation par *import*.

➥*import java.util.Scanner;* permet de nommer *Scanner* sans son préfixe *java.util*.

Quelques APIs importants du kit de développement J2SE (JDK) :

java.applet : contient des *interfaces* pour la création d'Applets (appliquettes) exécutable par un navigateur Web.

java.awt : Création de GUI (Préférer *javax.swing* dans les versions plus récentes de Java)

java.awt.event : Classes et interfaces pour la gestion des événements des packages *java.awt* et *javax.swing*

javax.swing : Classes et interfaces pour la création de GUIs (plus portables que awt/swt),

javax.swing.event : Classes et interfaces pour la gestion des événements dans les GUIs (par exemple le traitement des clics de souris, ...) et autres éléments du package *swing*.

java.io : Les entreées Sorties

java.lang : Classes et interfaces pour la plupart des programmes Java. Package importé par défaut.

java.net : Package pour les réseaux, internet...

java.util : Classes et interfaces utilitaires pour manipuler par exemple les dates, des nombres aléatoires, stockage et traitement d'une grande quantité de données, tokenization des strings, ...

java.text Classes et interfaces pour manipuler des nombres, dates, caractères, string, etc..., d'internatioalisation, ...

• D'autres packages permettent les manipulations avancées en graphiques, GUI plus sophistiqués, impression, réseaux avancés, sécurité, BDs, multimédia, accessibilité, etc.

0.26.1 Un exmple d'utilisation: les nombres aléatoires

• classe "*random*" de *java.util* ou la méthode static *Math.random()*

➥ La classe *random* de *java.util* perment de générer des valeurs aléatoires boolean, byte, short, int, long, float, double et Gaussienne.

➥ *Math.random* ne génère que des valeurs réelles $\in [0.0..1.0[$

• Création d'un objet générateur de nombres aléatoires :

```
Random nombresAleatoires = new Random();
```
avec *seed* entre () éventuellement

• L'obtention d'un entier (double, float, long, boolean, byte possibles)

```
int valeur = nombresAleatoires.nextInt();
```

➥ Les nombres (pseudo) aléatoires utilisent l'*heure* (écoulée depuis 1971 !) pour le *seed* (la graine).

• Pour obtenir 0 ou 1 : `int valeur = nombresAleatoires.nextInt( 2 );`

- Pour une valeur $\in 1..6$: `int valeur = 1+nombresAleatoires.nextInt( 6 );`

```
2
import java.util.Random;
4 public class Randoms
{
6    public static void main( String args[] )
     {
8       Random randomNumbers = new Random();
        int face;
10
        for ( int counter = 1; counter <= 20; counter++ )
12      {

14         face = 1 + randomNumbers.nextInt( 6 );
           System.out.printf( "%d   ", face );
16
           if ( counter % 5 == 0 )
18            System.out.println();
        }
20   }
}
```

- **Exécution :**

```
6    2    3    1    2
6    6    2    4    5
5    5    2    4    6
5    3    6    6    5
```

0.27 Un jeu avec des nombres aléatoires

- Réaliser le jeu (*craps*) suivant à l'aide de 2 dès lancés :

  - On gagne si la somme des 2 faces = 7 ou 11

  - On perd si la somme des 2 faces = 2, 3 ou 12

  - Si la somme des 2 faces est 4, 5, 6, 8, 9, 10 alors on a des "points" et on relance les dès jusqu'à obtenir le même "point" (même somme). On perdra si on fait 7 avant de refaire le "point".

- Eléments Java : utiliser des constates et "case", type énuméré (comme en C/C++)

```
1
3
5  import java.util.Random;
   class Craps
7  {   private Random aleatoires = new Random();
9      private  enum etat { Continuer, Gagnant, Perdant };
11     private final static int Oeil_Serpent = 2;
       private final static int trois = 3;
```

```
13    private final static int Sept = 7;
      private final static int Onze = 11;
15    private final static int Double_Six = 12;

17    public void jouer()
      {
19       int monPoint = 0;
         etat etatJeu;
21       int sommeDes = lancerDes();

23       switch ( sommeDes )
         {
25          case Sept:
            case Onze:
27             etatJeu = etat.Gagnant;
               break;
29          case Oeil_Serpent:
            case trois:
31    case Double_Six:
         etatJeu = etat.Perdant;
33       break;
      default:
35       etatJeu = etat.Continuer;
         monPoint = sommeDes;
37       System.out.printf( "Point = %d\n", monPoint );
         break;
39 }
         while ( etatJeu == etat.Continuer )
41       {
            sommeDes = lancerDes();
43
            if ( sommeDes == monPoint )
45             etatJeu = etat.Gagnant;
            else
```

```
47              if ( sommeDes == Sept )
                   etatJeu = etat.Perdant;
49          }

51          if ( etatJeu == etat.Gagnant )
               System.out.println( "==> Gagnant" );
53          else
               System.out.println( "==> Perdant" );
55      }

57      public int lancerDes()
        {
59          int des1 = 1 + aleatoires.nextInt( 6 );
            int des2 = 1 + aleatoires.nextInt( 6 );
61          int somme = des1 + des2;

63          System.out.printf( "Gagnant a lance %d + %d = %d\n",des1, des2, somme );
            return somme;
65      }
}
67


69
public class CrapsTest
71{
    public static void main( String args[] )
73  {
        Craps game = new Craps();
75      game.jouer();
    }
77}
```

- **Exécution (3 essais) :**

```
 Gagnant a lancé 1 + 6 = 7
 ==> Gagnant

Gagnant a lancé 4 + 1 = 5
Point = 5
Gagnant a lancé 5 + 6 = 11
Gagnant a lancé 6 + 4 = 10
Gagnant a lancé 4 + 3 = 7
==> Perdant

Gagnant a lancé 6 + 4 = 10
Point = 10
Gagnant a lancé 3 + 5 = 8
Gagnant a lancé 3 + 6 = 9
Gagnant a lancé 1 + 2 = 3
Gagnant a lancé 1 + 3 = 4
Gagnant a lancé 2 + 2 = 4
Gagnant a lancé 4 + 5 = 9
Gagnant a lancé 5 + 5 = 10
==> Gagnant
```

Remarques sur le code Java :

- Le type énuméré **etat** est déclaré private car on ne l'utilise que dans cette classe.

  ➥ Un type énuméré s'utilise comme une classe (notation avec '.'), private/public/etc.

  ➥ Par convention, les valeurs (uniques) d'un type énuméré commencent par un majuscule.

- Les onstantes sont static pour éviter que les instances les répètent (inutile).

  ➥ lle type énuméré aurait pu être *static* pour la même raison.

- La classe public contenant *main* pourrait être dans un fichier séparé.

N.B. : Porté des variables comme en C/C++

➥ mais pas possible de déclarer une variable locale idem qu'un argument (?).

0.28 Surcharge

• Signature d'une fonction = le nombre, les types ou l'ordre différents pour les paramètres d'une méthode.

➥ Le compilateur utilise la signature en interne (dans l'ordre) : nom, nombre, types, ordre.

➥ Le type de retour n'est pas utilisé (non discriminant)

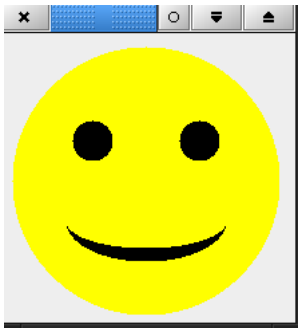
➥ Exemple (Math) : abs, min et max sont surchargées et différenciées par des types double, float, long et int.

0.29 Un autre exemple graphique : type énuméré Color (couleur)

```
1
2
3  import java.awt.Color;
   import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7
   import javax.swing.JFrame;
9
   class DrawSmiley extends JPanel
11 {
       public void paintComponent( Graphics g )
13     {
           super.paintComponent( g );
15
           g.setColor( Color.YELLOW );
17         g.fillOval( 10, 10, 200, 200 );
18
19         g.setColor( Color.BLACK );
           g.fillOval( 55, 65, 30, 30 );
21         g.fillOval( 135, 65, 30, 30 );
22
23         g.fillOval( 50, 110, 120, 60 );
24
25         g.setColor( Color.YELLOW );
           g.fillRect( 50, 110, 120, 30 );
27         g.fillOval( 50, 120, 120, 40 );
28     }
29 }
30
31
```

```
33 public class SmileyTest
   {
35    public static void main( String args[] )
      {
37       DrawSmiley panel = new DrawSmiley();
         JFrame application = new JFrame();
39       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
         application.add( panel );
41       application.setSize( 230, 250 );
         application.setVisible( true );
43    }
   }
```

- **Exécution :**



0.30 Tableaux statiques avec new

- Déclaration : `int tab[] = new int[ 12 ];`

  ➥ Ou alternativement

  ➥ `int [] tab1,tab12; tab11 = new int[ 10 ];..` pour la mise en facteur

  ➥ Erreur de compilation : `int tab[ 12 ];`

- Initialisation : `int nombres [] = { 10, 20, 30, 40, 50 };`

- La taille connue à la création :

```
final int Taille= 10;                    //  constante
int array[] = new int[ Taille ];   // création
```

- L'accès illégale provoque l'exception : *ArrayIndexOutOfBoundsException*

- Exemple : Main de Pocker (fourni) avec des tableaux.

0.30.1 Recherche d'un éléments dans un Array (tableau)

• Recherche dans un tableau ordonné de String par une méthode Dichotomique:

```
String[] tabStringTrie = new String[]{"ant", "bat", "cat", "dog"};// Créat

  // Recherche de  "cat"
int index = Arrays.binarySearch(tabStringTrie, "cat");    // 2

  // Recherche d'un élément non existant
index = Arrays.binarySearch(tabStringTrie, "cow");        // -4=-taille
```

• Pour un type primitif :

```
int[] tabIntTrie = new int[]{1, 2, 3, 5, 7};    // création

  // Recherche de 6
index = Arrays.binarySearch(tabIntTrie, 6);         // -5
```

0.31 Type String (suite)

- **Comparaison de Strings :**

  égalite : **equals** (== compare les adresses)

  Pour les autres comparaisons : **compareTo()**

- Voir plus loin. Ici, quelques exemples

Exemples :

```
String s = "quelque chose", t = "autre chose";
if (s.equals(t))  // OK
if (s.compareTo(t) > 0) {..} // OK

int i = s.compareTo(t);      // Les Majuscules  plus petits
if (i < 0) {...}             // s plus petit  que t
else if (i > 0) {...}      // plus grand
      else {...}           // égaux
```

- Pour comparer sans tenir compte de la "case" : *compareToIgnoreCase*

Construction des strings :

```
StringBuffer buf = new StringBuffer("Java");

// Append
buf.append(" Almanac v1/");      // Java Almanac v1/
buf.append(3);                   // Java Almanac v1/3

// Set
int index = 15;
buf.setCharAt(index, '.');       // Java Almanac v1.3

// Insert
index = 5;
buf.insert(index, "Developers ");// Java Developers Almanac v1.3
```

Suite de l'exemple :

```
// Replace
int start = 27, end = 28;
buf.replace(start, end, "4");    // Java Developers Almanac v1.4

// Delete
start = 24;
end = 25;
buf.delete(start, end);          // Java Developers Almanac 1.4

// Convert to string
String s = buf.toString();
```

Présence de sous chaine :

```
String string = "Madam, Je suis Adam";

// Starts with (commence avec)
boolean  b = string.startsWith("Mad");  // Oui

// Ends with (chaine fini ave..)
b = string.endsWith("dam");             // OOui

// N'importe  où
b = string.indexOf("Je suis") > 0;      // Oui
```

Suite de l'exemple :

- On peut ignorer la "case" avec des expressions régulières (ER) où

➥ Le symbole " (?i) " prévient de l'utilisation d'une ER.

➥ " .* " veut dire : un nombre quelconque de n'importe quel caractère (rien ou tout !)

```
// ccommence avec :
b = string.matches("(?i)mad.*"); // "mad" au début suivi de n'importe

// Fini avec
b = string.matches("(?i).*adam"); // n'importe suivi de "adam"
     // avec  "adam" est suivi de rien

// N'importe où
b = string.matches("(?i).*Je suis.*"); // n'importe puis "Je suis"
    // puis n'importe
```

Recherche de sous chaine :

```
String string = "Madame, je suis Adam";

   // Caratères

    // Première occurrence d'un carac
    int index = string.indexOf('a');    // 1

    // dernière occurrence
    index = string.lastIndexOf('a');    // 16

    //Pas trouvé
    index = string.lastIndexOf('z');    // -1
```

Suite de l'exemple :

```
// Sous chaines

    // Première occurrence d'une sous chaine
    index = string.indexOf("dam");      // 2

    // dernière occurrence d'une sous chaine
    index = string.lastIndexOf("dam");  // 15

    // pas trouvé
    index = string.lastIndexOf("z");    // -1
```

Extraction d'une sous chaine :

```
int start = 1;
   int end = 4;
   String substr = "aString".substring(start, end);   // Str
```

Remplacement d'un car dans un string :

```
   // Remplacer tout 'a' par 'o'
   String newChaine = string.replace('a', 'o');
```

Remplacement d'une sous chaine dans un string :

A titre d'indication, le code Java de la fonction "*replace*" de la classe "*string*" du paquetage *java.lang* ressemble à :

```
static String replace(String str, String pattern, String replace) {
    int s = 0;
    int e = 0;
    StringBuffer result = new StringBuffer();

    while ((e = str.indexOf(pattern, s)) >= 0) {
        result.append(str.substring(s, e));
        result.append(replace);
        s = e+pattern.length();
    }
    result.append(str.substring(s));
    return result.toString();
}
```

convesrion MIN/MAJ :

```
// Convertir en Maj
String upper = string.toUpperCase();

// Convertir en Min
String lower = string.toLowerCase();
```

convesrion des types primitifs au string :

```
// Utiliser  String.valueOf()
   String s = String.valueOf(true);      // true
   s = String.valueOf((byte)0x12);       // 18
   s = String.valueOf((byte)0xFF);       // -1
   s = String.valueOf('a');              // a
   s = String.valueOf((short)123);       // 123
   s = String.valueOf(123);              // 123
   s = String.valueOf(123L);             // 123
   s = String.valueOf(1.23F);            // 1.23
   s = String.valueOf(1.23D);            // 1.23

   // Utiliser  +
   s = ""+true;                          // true
   s = ""+((byte)0x12);                  // 18
   s = ""+((byte)0xFF);                  // -1
   s = ""+'a';                           // a
   s = ""+((short)123);                  // 123
   s = ""+123;                           // 123
   s = ""+123L;                          // 123
   s = ""+1.23F;                         // 1.23
   s = ""+1.23D;                         // 1.23
```

0.32 Généricté en Java

0.32.1 Exemple des nombres premiers revisité (vector<...>)

```
2

4   import java.util.*;
    import java.lang.Math;
6

public class PrimeGenerics {
8    static Vector<Integer> vect;
```

```
10    public static void main(String[] args) {
          java.util.Scanner in = new java.util.Scanner(System.in);
12        System.out.println("quel est le max > 2 ?");
          int valeur= in.nextInt();
14        lesPrimesAvecVect(valeur);System.out.println();

16    }

18    public  static void lesPrimesAvecVect(int val) {

20         vect=new Vector<Integer>();
           vect.add(1);vect.add(2);
22         for (int i=3; i<= val; i+=2)
              {
24             if (! possedUnMultipleDansVect(i, (int)(java.lang.Math.sqrt(i))))
                    {  vect.add(i);
26
                }
```

```
28              }
        System.out.printf("\n Les résultats \n ");
30      System.out.println(vect.toString());
    }
32
    public static boolean possedUnMultipleDansVect(int val, int limite) {
34          for (int i=2; i < vect.size(); i++)
                {if (val %  vect.elementAt(i) ==0 ) return true;
36               if ( vect.get(i) > limite) return false;
                }
38      return false;
        }
40
    }
```

- **Exécution :**

```
quel est le max > 2 ?
12

 Les résultats
 [1, 2, 3, 5, 7, 11]
```

0.33 Java vs. C++

- Selon certains, Java a été inventé pour simplifier et nettoyer C++.
- Les deux syntaxiquement très proches;
- Mais il y a des différences fondamentales sous jacentes.
- Les commentaires sont identiques dans les deux langages;
- Un exemple simple en Java (discuter la version C++) :

```
2 public class java_vs_cpp1
{
4  private int factorielle(int n)
   {  int i;
6     int answer = 1;
      for(i=2;i<=n;i++)
8       answer = answer * i;
        return answer;
10 }

12 private int factorielle_rec(int n)
   {  if (n<2) return 1;
14     else return n*factorielle_rec(n-1);
   }
16

   public static void main(String[] args)
```

```
18  { java_vs_cpp1 prog = new java_vs_cpp1();
    int m = 5;
20  System.out.println("La factorielle de " + m + " est " + prog.factorielle(m));
    System.out.println("Avec la version récursive :  " + prog.factorielle_rec(m));
22  }

24 }
```

- **Exécution :**

```
La factorielle de 5 est 120
Avec la version récursive :  120
```

0.33.1 Remarques sur l'exemple java_vs_cpp1.java

• En Java, tout est dans une classe.

➥ C++ laisse un choix à ce niveau, pas Java !

• Java est plus proche du paradigme Orienté Objets que C++ (qui est une évolution de C).

• Il peut y avoir plus d'une classe dans le même fichier Java mais seule une classe peut être **public**.

➧ Le fichier contenant le code doit avoir le même nom (et l'extension *.java*) que cette classe *public*.

➧ La classe *public* contient une méthode **main** (le point d'entrée) qui doit être déclarée par :

public static void.

• En Java, l'argument de la fonction *main* est un tableau de *string* : main(String[] args) dont la taille est disponible (*length*).

◗ En C++, main a deux arguements optionnels de la forme main(int nbArgs, char* args[])) (*nbArgs* est nécessaire).

• Les attributs et les méthodes d'une classe peuvent être "public", "private" (ou "protected").

◗ La signification de ces 3 labels est identique dans les 2 langages.

◗ C++ fait de même mais permet à une section entière de porter le même label (e.g. "public"),

➥ Java demande à répéter ce label.

➥ Mais si un label explicite est oublié, la valeur par défaut "friendly" = amical est supposée (voir plus bas).

• Le sens des fonctions (factorielles) de l'exemple reste identique dans les 2 langages.

• Du fait de devoir définir "main" dans une classe, il faut déclarer une instance de la classe public *java_vs_cpp1* avant de pouvoir appeler l'une de ses méthodes (factorielles).

◗ On peut éviter cela en déclarant la classe *static*.

◗ On remarque que ces appels ont lieu sur l'objet *prog*.

- Les entrées sorties simples en Java avec **System.out.println** (ou *printf*)
  - L'équivalent C++ (sans classe) serait :

```
cout « "La factorielle de " « m « " est " « factorielle(m) « endl;
```

  - Le symbole"+" en Java permet la concatenation de String.
  - Java convertit automatiquement les nombres (cf. *m*) en un string imprimable (comme *cout*).
  - **System.out.print** écrit sans passer à la ligne.
- Remarquez aussi que l'accolade de fin d'une classe n'a pas de ";" (il le faut en C++).

0.33.2 *Les objets en java*

- L'exemple suivant montre d'autres différences importantes entre Java et C++.
- Exemple de compte bancaire :

```
2  public class java_vs_cpp_compte
   {
4      public static void main(String[] args)
       {   compte moncompte = new compte(200.0,1);
6          compte toncompte = new compte(300.0,2);

8          moncompte.retrait(30.0);
           toncompte.depot(40.0);
10          moncompte.etat(); toncompte.etat();
       }
12 }

14 class compte
   {
16     private double solde;
       private int compteId;
18
       public compte(double b, int id )
20     { solde = b;
         compteId = id;
22     }

24     public void depot(double montant)
       { solde = solde + montant; }
```

```
26
     public void retrait(double montant)
28   { if (montant >= solde) solde = solde - montant; }

30   public void etat()
     { System.out.println("Solde du compte " + compteId + " : " + solde);
32   }
}
34
```

- **Exécution :**

```
Solde du compte 1 : 200.0
Solde du compte 2 : 340.0
```

0.33.3 Remarques sur l'exemple java_vs_cpp_compte.java

• Il y a deux classes dans ce code, mais une seule est *public*

➥ *compte* sans label = *friendly*

➥ Un élément est *friendly* par défaut. Il est accessible uniquement aux classes de son package (et à ses classes filles).

➥ Ceci signifie qu'un objet de cette classe peut être créé par toute autre classe du package, mais pas en dehors du package

➥ Il ne peut y avoir qu'une seule classe public par unité de compilation (fichier).

➥ L'idée est que chaque unité de compilation ait une seule interface publique représentée par cette classe *public* . Elle peut avoir autant de classes "amicales" utilitaires qu'on veut.

➥ Si on a plus d'une classe *public* dans une unité de compilation, le compilateur générera un message d'erreur.

• Dans l'exemple, la classe *compte* est (de préférence) définie **après** la classe *public* : ce qui n'est pas possible en C++ sans un prototype.

➥ Java permet cette pré-déclaration (référence en amont = forward referencing) pour des classes mais aussi pour des attributs et fonctions permettant une analyse descendant.

• Autre différence : tous les objets Java sont des références (pointeurs). Voir types primitifs.

➧ Dans le programme ci-dessus, l'expression *moncompte.retrait(30.0);* aurait été écrite en C++ par *moncompte->retrait(30.0);*. En java, la notation est **invariable**.

➧ En Java, tous les types de base (int, char, double, ...) sont des variables directes (non-pointers).

➧ En Java, tous types complexes (y compris les tableaux, strings) et les instances des classes doivent être définies par référence (avec *new*).

➥ C'est une convention fixe : il n'y a pas de "&" devant (par exemple un *int*) pour en faire un pointeur (c'est le cas en C++).

➥ Pas de * ni **->** pour dire qu'on a un pointeur : ils sont implicites et nécessitent "new".

➥ **A.B** de Java correspond à **A->B** de C++ dans ce cas.

➥ **A.B** de C++ n'a pas d'équivalent en Java !

• L'utilisation des pointeurs en Java est plus stricte.

Résumé :

✗ Les types primitifs de Java sont non-pointeurs et toutes les variables instances sont des pointeurs.

✗ Il n'y a pas de "->", "*" ou "&" en java.

✗ "A.B" de C++ correspond à "A->B" de C++.

✗ Le "A.B" n'a pas d'équivalent en Java.

0.33.4 Héritage en Java

- Syntaxe Java :

```
class compte_epargne extends compte { ... }
```

  ➥ En C++ :

```
class compte_epargne : compte { ... };
```

- Définition d'une méthode pour une classe en dehors de la classe :

  ➥ En C++, on définit par `nom_classe::nom_méthode() {...}`

  ➥ En Java, ceci n'existe pas.

- Autre différence : en Java, il n'y a pas d'héritage multiple.... (existe en C++).

0.33.5 Destructeur en Java

• Java incorpore un ramasseur de miettes.

• La machine virtuelle Java réclame automatiquement de la mémoire pour les objets qui ne sont plus référencés.

➥ Il n'y a donc pas besoin de destructeur (ni delete, ni free), comme c'est le cas en C++.

0.33.6 Les types (saufs)

• Une amélioration importante de Java par rapport à C++ est la sécurité de type.

• Par exemple, la valeur d'un tableau de caractères ne peut devenir un entier.

• Ceci pose problème en C++ (car un pointeur est représenté par un entier):

```
int A[4];          // voir ci-dessous pour les tableaux Java
int x = 0;
if (A>x) x++;      // Pas bon en Java
if ( (int) A > x ) x++;  // Pas bon en java
```

• On ne peut pas comparer en Java u tableau et un entier (en fait, A est l'adresse du début de tableau).

• En Java, même la conversion de types entre certains types est interdite.

• Typage fort est mieux (plus sur). Le langage C# a aussi adopté cela.

• Les types Java *Booleans*, *arrays* (tableaux), et *strings* montrent aussi un meilleur traitement des types en Java.

➥ Boolean : il y a les constantes true et false; pas 0 et non-zéro (comme en C++).

• Par exemple, l'expression C++ (étrange) :

```
while (4 && 5)  cout « "4 et 5 sont tous deux vrais!";
```

• Les opérateurs booléens (&&, ||, etc.) sont les mêmes qu'en C++.

• Le code Java ci-dessus montre exemple (c'est aussi un code C++) :

```
boolean A = true;
boolean B = false;
while (A || B) { A = !B;  B = A && B; }  // Bouvle infinie ?
```

• Tableaux (arrays) en Java :

```
int[] monTableau;
monTableau = new int[5];
```

ou

```
int monTableau[] = new int[5];
```

➥ On déclare ici un tableau de 5 entiers.

➥ Les tableaux sont des objets Java.

➥ Ces objets ont leurs membres et méthodes :

Par exemple : 'monTableau.length' contient la taille du tableau (ici 5).

➥ A[i] représente comme en C++ le i+1ème.

• Les tableaux sont des objets : le tableau *monTableau* est un pointeur.

➥ Si l'on le passe à une fonction qui le modifie, les modifications seront répercutées dans l'appelant (comme en C++).

Rappel : en Java il n'y a pas de "*", par exemple **(monTableau+2)* comme en C++ :

➥ on accède aux éléments d'un tableau par un indice comme *monTableau[2]*.

• Tableaux multi-dimentionnels :

```
int[][] matrice = new int[5][5];
```

• Chaîne de caractères (String) :

L'expression Java `String salut = "hello";`

Correspond à l'expression C++ `char salut[] = "hello";`

• Un string Java ne doit pas être considéré comme un pointeur sur caractères (char *).

• Comme les tableaux, les Strings sont des objets et ont des méthodes.

➥ Par exemple, 'salut.length()' renvoie la longueur de cette chaine (5 ici).

➥ Le symbole '+' est utilisé pour la concaténation :

➥ ("abc" + "def") représente le string "abcdef".

• Puisuq'en Java, tous les objets sont des pointeurs, on ne peut pas utiliser '==' pour tester l'égalité des chaine ('==' testerait l'égalité des pointeurs).

➥ La classe String de Java contient la méthode "equals" pour cela.

➥ Par exemple,

`salut.equals("hello")` qui renvoie vraie. Ici, on a une comparaison d'objets.

0.33.7 Les flots d'entrées/sorties

• En Java, les entrées/sorties avec un clavier, un fichier disque ou une socket TCP/IP passent par des flots (*streams*).

• Exemple pour le clavier :

◗ 1. insérer la ligne

```
import java.io.*;
```

"import" est comme "include" en C/C++. Cette ligne inclue tout de la librairie **java.io library**.

◗ 2. Pour utiliser les E/S standard, on doit utiliser le bloc "try-catch" pour récupérer les exceptions.

➥ A l'intérieur, on définit un objet "BufferedReader".

Ensuite, on procède à la lecture par la méthode "readline" de cet objet.

Exemple :

```
try {
   String uneChaine;      // contiendra la chaine
   int unEntier;          // un entier à lire
   BufferedReader BR;    // opour lire depuis le clavier
   BR = new BufferedReader( new InputStreamReader(System.in) );
   uneChaine = BR.readLine();  // relire une ligne du clavier
   // conversion   string --> entier : par exemple   "12" --> 12:
   unEntier = Integer.parseInt(uneChaine);
   BR.close();  // fermer BufferedReader
} catch (Exception E) {System.out.println(E);}
```

• L'objet "BR" de la classe *BufferedReader* correspond à **cin** de C++. (voir aussi *Scanner*).

• Cet objet possède plusieurs méthodes dont *readLine* utilisée pour lire.

➥ On peut convertir la chaîne lue en d'autres types.

➥ Par exemple, Integer.parseInt("12") renverra l'entier 12.

• Le complément en sortie à *BufferedReader* est "PrintWriter":

```
PrintWriter PR;
PR = new PrintWriter(new OutputStreamWriter(System.out));
PR.println("hello");
```

• On placera ces ligne dans un bloc *try-catch*

➥ et ce sera équivalent à `System.out.println("hello");`.

• Le même mécanisme est utilisé pour les autres flots :

```
BR = new BufferedReader(new InputStreamReader
     (new FileInputStream("monfic")));
PR = new PrintWriter(new OutputStreamWriter
                              (new FileOutputStream("monfic")));
```

• Cela permettra aux objets BR et PR de lire depuis le fichier "monfic".

➥ 'readLine' lira des lignes ASCII et 'println' les écrira.

• Exemple Socket :

```
import java.net.*;
     ...
Socket www = new Socket(InetAddress.getByName("www.hofstra.edu"),80);
BR = new BufferedReader(new InputStreamReader(www.getInputStream()));
PR = new PrintWriter(new OutputStreamWriter(www.getOutputStream()),true);
```

Ceci permet la communication avec le serveur WEB *hofstra* par les mêmes 'readLine' et 'println'.

• "BR.close();" et "PR.close();" sont utilisées pour fermer les flots.

• Voir le programme ci-dessus aussi.

N.B. : il y divers packages développés pour les entrées sorties au clavier.

➥ Le type Scanner de Java convient bien.

➥ Voir aussi www.cs.hofstra.edu/java/Console.java. A placer dans le répertoire d'utilisation.

➥ Un exemple d'utilisation est donné ci-dessous.

Autres remarques sur la différence Java/C++ :

- Le pointeur NULL n'est pas équivalent à 0 en Java
- La valeur "null" représente un pointeur nul.
- Les variables de type primitif (e.g. int) sont initialisées (0 pour un entier).

➥ Par contre , une telle variable déclarée à l'intérieur d'une méthode doit être initialisée (C++ ne dit rien à ce sujet).

- De même, dans une déclaration telle que "Compte cpt";
  - ➥ *cpt* est initialisée à "null" (puisque référence) s'il s'agit d'un membre d'une classe.
  - ➥ Par contre, ele doit être initialisée si elle est déclarée à l'intérieur d'une méthode.

0.33.8 Un exemple récapitulatif : lecture d'une liste d'entiers

```
1
3  import java.io.*;

5  class cell
   {
7     int head;
      cell tail;
9
   public cell(int h, cell t)
11      { head = h;  tail = t; }


13
   public void print()
15      {  cell ptr = this;
           while (ptr != null)
17         {
               System.out.print(ptr.head + " ");
19             ptr = ptr.tail;
               }
21         System.out.println("");
        }
23 }

25 public class ex_liste
   {
27
     public static void main(String[] args)
29      { int i;
          String input;
31        int A[] = new int[5];
```

```
        cell L = null;
33      try {

35            BufferedReader BR;
              BR = new BufferedReader( new InputStreamReader(System.in) );
37        for(i=0;i<A.length;i++)
          {  System.out.print(" Entrer un entier: ");
39           input = BR.readLine();
                     A[i] = Integer.parseInt(input);
41        }

43            for(i=A.length-1;i>=0;i--)
              L = new cell(A[i],L);
45
              System.out.println("Contenu de la liste :");
47            L.print();
              BR.close();
49          }  catch (Exception E) { System.out.println(E); }

51    }
```

- **Exécution :**

```
 Entrer un entier: 1
 Entrer un entier: 2
 Entrer un entier: 3
 Entrer un entier: 4
 Entrer un entier: 5
Contenu de la liste :
1 2 3 4 5
```

Donné aux élèves jusqu'à la

0.33.9 Les niveaux d'accès en Java

✓ Java définit quatre niveaux d'accès pour les variables d'instances et les méthodes :

- **public** : un élément public est accessible de partout et sans aucune restriction.

  Certaines classes (comme la classe principale *main*) doivent obligatoirement être déclarées publiques (pour pouvoir exécuter l'application...)

- **protected** : un élément **protected** (protégé) est accessible uniquement aux classes d'un package et à ses classes filles

- **private** : un élément *private* (privé) est accessible uniquement au sein de la classe dans laquelle il est déclaré.

  ➥ Ces éléments ne peuvent être manipulés qu'à l'aide de méthode spécifiques appelés *accesseur* (*getxxx(...)*) et *mutateur* (*setxxx()*) pour la variable xxx.

◗ **friendly** : un élément est *friendly* par défaut (cette appellation n'est pas officielle et est empruntée au langage C++) est accessible uniquement aux classes d'un package et à ses classes filles.

➥ Il est possible, bien que non habituel, d'avoir une unité de compilation sans aucune classe public. Dans ce cas, on peut appeler le fichier comme on veut.

➥ La classe public peut avoir besoin d'autres classes (utilitaires) dont les détails seront cachés aux utilisateurs (on pourra changer leur implantation, voire leur nom à notre guise).

➥ Pour réaliser ceci il suffit d'enlever le mot-clé public de la classe, qui devient dans ce cas amicale. (Cette classe ne peut être utilisée que dans ce package.)

• Remarquez : une classe ne peut pas être private (cela ne la rendrait accessible à personne d'autre que cette classe), ou protected.

➥ Il n'y a donc que deux choix pour l'accès aux classes : « amical » ou public.

➥ Si on ne veut pas que quelqu'un d'autre accède à cette classe, on peut rendre tous les

constructeurs private, ce qui empêche tout le monde de créer un objet de cette classe, à part soi-même dans un membre static de la classe.

➥ Comme mentionné précédemment, si on ne met pas de spécificateur d'accès il est « amical » par défaut.

➥ Ceci signifie qu'un objet de cette classe peut être créé par toute autre classe du package, mais pas en dehors du package (souvenez-vous que tous les fichiers dans le même répertoire qui n'ont pas de déclaration package explicite font implicitement partie du package par défaut pour ce répertoire).

➥ Cependant, si un membre static de cette classe est public, le programmeur client peut encore accéder à ce membre static même s'il ne peut pas créer un objet de cette classe.

0.33.10 Un exemple d'accès

- Cet exemple montre les spécificateurs d'accès de classes.

- Faire une classe effectivement private avec des constructeurs private :

```
class Soup {
  private Soup() {}
  // (1) Permettre la création à l'aide d'une méthode static :
  public static Soup makeSoup() {    return new Soup();  }
  // (2) Créer un objet static et retourner une référence à la demande.
  // (le patron "Singleton"):
  private static Soup ps1 = new Soup();
  public static Soup access() {    return ps1;  }
  public void f() {}
}
class Sandwich { // Utilise Lunch
  void f() { new Lunch(); }
}
// Une seule classe public autorisée par fichier :
public class Lunch {
  void test() {
// Ne peut pas faire ceci ! Constructeur privé :
    //! Soup priv1 = new Soup();
    Soup priv2 = Soup.makeSoup();
    Sandwich f1 = new Sandwich();
    Soup.access().f();
  }
}
```

Cette méthode retourne une référence à un objet de la classe Soup.

```
public static Soup access() {    return ps1;  }
```

La classe Soup montre comment empêcher la création directe d'une classe en rendant tous les constructeurs private.

Rappel : si vous ne créez pas explicitement au moins un constructeur, le constructeur par défaut (un constructeur sans arguments) sera créé pour vous.

➥ En écrivant ce constructeur par défaut, il ne sera pas créé automatiquement.

➥ En le rendant private, personne ne pourra créer un objet de cette classe.

➥ Mais alors comment utilise-t-on cette classe ?

➥ L'exemple ci-dessus montre deux possibilités.

➤ Premièrement, une méthode static est créée, elle créee un nouveau Soup et en retourne la référence. Ceci peut être utile si on veut faire des opérations supplémentaires sur Soup avant de

le retourner, ou si on veut garder un compteur du nombre d'objets Soup créés (peut-être pour restreindre leur population).

➤ La seconde possibilité utilise ce qu'on appelle un patron de conception [design pattern], qui est expliqué dans Thinking in Patterns with Java, téléchargeable sur www.BruceEckel.com.

Ce patron particulier est appelé un "singleton" parce qu'il n'autorise la création que d'un seul objet.

L'objet de classe Soup est créé comme un membre static private de Soup, ce qui fait qu'il n'y en a qu'un seul, et on ne peut y accéder qu'à travers la méthode public access().

0.34 Passer des versions préc de Java à la nouvelle

• Java 5 (JDK 1.5) a apporté bcp de nouvelles choses.

Les remplacements à faire :

• Replace **StringBuffer** with **StringBuilder**

The older StringBuffer class is thread-safe, while the new StringBuilder is not. However, it's almost always used in a context in which thread safety is superfluous. Hence, the extra cost of synchronization is paid without benefit.

Although the performance improvement in moving from StringBuffer to StringBuilder may only be very slight (or perhaps not even measurable), it's generally considered better form to prefer StringBuilder.

• Use sequence parameters when appropriate

Sequence parameters (varargs) let you replace Object[] parameters (containing 0..N items) appearing at the end of a parameter list with an alternate form more convenient for the caller. For

example,

public static void main(String[] aArgs)

can now be replaced with :

public static void main(String... aArgs)

• Replace constants with enumerations

Using public static final constants to represent sets of related items should be avoided in favor of the enumeration classes now supported by Java. In addition, you should consider replacing any "roll-your-own" implementations of type-safe enumerations with the new language construct.

• Use @Override liberally

The @Override standard annotation identifies methods that override a superclass method. It should be used liberally to indicate your intent to override.

• Avoid raw types (i.e. generic est mieux)

Raw types should almost always be avoided in favor of parameterized types.

• Use for-each loops

The enhanced for loop (also called the for-each loop) should be used whenever available. It is more compact, concise, and clear.

• Be careful with Comparable

The Comparable interface has been made generic. For example,

```
class Anatomy implements Comparable{
  public int compareTo(Object aThat){}
}
```

should now be replaced with :

```
class Anatomy implements Comparable<Anatomy>{
  public int compareTo(Anatomy aThat){}
}
```

Example

Here is an example of code written using Java 5 features :

1

```
23
25
27
29
31
33
35
37
39
41
43
45
47
49 import java.util.*;
51 public final class test_java15 {
53
```

```
55
57
59  public static void main(String... aArgs){
61    List<String> lEmployes = Arrays.asList("Thomas", "Fiorella", "Pedro");
      test_java15 bureau = new test_java15(AirConditionne.COUPE, lEmployes);
63    System.out.println(bureau);
65
      for(String tournee: lEmployes){
67      System.out.println(tournee);
      }
69  }
71
73
    enum AirConditionne {COUPE, FAIBLE, MOYEN, FORT}
75
77
79  public static final int COUPE = 1;
    public static final int FAIBLE = 2;
81  public static final int MOYEN = 3;
    public static final int FORT = 4;
83
    test_java15(AirConditionne unAirConditionne, List<String> desEmployes){
85    fAirConditionne = unAirConditionne;
      fEmployes = desEmployes;
87  }
```

```
89    AirConditionne getAirConditionne(){
        return fAirConditionne;
91    }

93    List<String> getEmployes(){
        return fEmployes;
95    }

97



99



101   @Override public String toString(){

103     return("toString à faire" + fEmployes.toString());
      }
105
      @Override public boolean equals(Object unAutre){
107
        return true;
109   }

111   @Override public int hashCode(){

113     return 0;
      }
115


117   private final List<String> fEmployes;
      private final AirConditionne fAirConditionne;
119 }

121
```

```
123

125
```

- **Exécution :**

```
toString à faire[Thomas, Fiorella, Pedro]
Thomas
Fiorella
Pedro
```

0.35 Type énuméré (suite) en JDK 1.5 et +

• Est un ensemble de valeurs normalement liées.

Par exemple :

- Des directions : nord, sud, ...
- Genres livres : roman, fantaisie, fantastique, classique, ...
- Types de glaces : crème, chocolat, vanille, ...

• On les appelle aussi *Type-safe enumerations* ou *enums*, ont été ajoutés à JDK 1.5 comme une sorte de classe particuliée.

• Si non disponible dans une version antérieure de Java, on peut les implanter comme une classe régulière.

• On peut s'en servir à l'occasion à la place des constantes de type entier, string, etc.

• Un type *enum* est implicitement sous classe de *java.lang.Enum*

• Si un *enum* est membre d'une classe, il est implicitement *static*

• Jamais de *new* avec un enum , même à l'intérieur du type énuméré lui même

• Les méthodes **name** et **valueOf** utilsent le texte des constantes énumérées alors que la méthode **toString** peut être surchargée au besoin (voir l'exemple suivant).

• Pour les constantes énumérées, *equals* et == reviennent au même (interchangeables)

• Les constantes énumérées sont implicitement *public static final*

• L'ordre apparente des constantes énumérées appélé "ordre naturel" est celui utilisé par les méthodes *compareTo*, dans une itération sur leur valeur, dans *EnumSet* et dans *EnumSet.range*.

• On peut définir n'importe quelle méthode dans un type énuméré pour modifier l'état d'une constante énumérée.

➥ Donc, le terme "costante énumérée" est ambigue.

➥ Ce qui est en fait constante est la valeur d'un élément énuméré, pas son état. Il faut plutôt utiliser le terme "élément énuméré" et non "constante énumérée".

• Un constructeur pour un type énuméré doit être déclaré *private*. Le compilateur permet la

déclaration d'un constructeur non private mais ce sera pas clair car new est interdit dans un type énum.

• Exemples :

➥ On remarque la nouvelle forme de la boucle *for*.

➥ On remarque aussi *EnumSet*<...>

A propos des **Quarks** :

➥ Sont des prticules élémentaires, constituant principal de la matière.

➥ Six sortes différentes : UP, DOWN, CHERM, STRANGE, TOP, BOTTOM

➥ UP & DOWN : masse minimum, plus stables, très commun dans l'univers

➥ Les autres sont plus "massifs", se transforment (se réduisent) plus rapidement vers les types UP & DOWN

➥ A cause de cela, ces 4 autres Quarks sont produits par collision dans les milieux (accéléra-

teurs) chargés d'une grande quantité d'énergie .

```
1
import java.util.EnumSet;
3
public final class EnumExamples {
5
  public static final void main(String... lesArgsVariables){
7   log("Exercice dsur les énumerations...");
    testEnumsDivers();
9   testMutableEnum();
    testEnumRange();
11  testBitFlags();
    testEnumToStringEtTestValueOf();
13  log("Fini.");
  }
15

17 private static void log(Object unTexte){
    System.out.println(String.valueOf(unTexte));
19 }

21
  enum Quark {
23



25



27



29



31    UP,
      DOWN,
33    CHARM,
```

```
    STRANGE,
35  BOTTOM,
    TOP
37 }

39

41

43

45

47

49
   public enum Lepan {
51
    ELECTRON(-1, 1.0E-31),
53  NEUTRINO(0, 0.0);

55

57

59

61
    private Lepan(int uneCharge, double uneMasse){
63

      nouvCharge = uneCharge;
65    nouvMasse = uneMasse;
67  }
```

```
    final int getCharge() {
69    return nouvCharge;
    }
71  final double getMasse() {
      return nouvMasse;
73  }
    private final int nouvCharge;
75  private final double nouvMasse;
}
77


79


81


83
static enum Direction {
85  NORD,
    SUD,
87  EST,
    OEUST;
89


91
    @Override public String toString(){
93


95


97


99    return "Direction: " + name();
    }
101
```

```
    public boolean estFroid() {
103
      return  this == NORD;
105   }
  }
107


109


111 private enum Parfum {
    CHOCOLAT(100),
113 VANILLE(120),
    CERISE(80);
115
    void setCalories(int desCalories){
117
      fCalories = desCalories;
119   }
    int getCalories(){
121   return fCalories;
    }
123 private Parfum(int desCalories){
      fCalories = desCalories;
125   }


127
    private int fCalories;
129 }

131


133


135 private static void testEnumsDivers(){
```

```
137     log("toString: " + Quark.BOTTOM);

139
        if ( Quark.UP == Quark.UP ) {
141       log("UP == UP");
        }
143
        if ( Quark.UP.equals(Quark.UP) ) {
145       log("UP.equals(UP) !! ");
        }
147


149
        if ( Quark.UP.compareTo(Quark.DOWN) < 0 ) {
151
          log("UP avant DOWN");
153     }
        else if ( Quark.UP.compareTo(Quark.DOWN) > 0 ) {
155       log("DOWN avant UP");
        }
157     else {
          log("UP comme DOWN");
159     }

161
        log("Valeurs de Quark : " + Quark.values());
163
        for ( Quark quark : Quark.values() ){
165       log("Item dans Quark.values() : " + quark);
        }
167
        log("toString : " + Direction.NORD);
169     if ( Direction.EST.estFroid() ){
```

```
      log("Est est froid");
171   }
      else {
173     log("Est n'est pas froid.");
      }
175
      log("Electron charge : " + Lepan.ELECTRON.getCharge());
177

179   Lepan lepan = Enum.valueOf(Lepan.class, "ELECTRON");
      log("Masse de Lepan : " + lepan.getMasse());
181

183   try {
        Lepan unAutreLepan = Enum.valueOf(Lepan.class, "Proan");
185   }
      catch (IllegalArgumentException ex){
187     log("Proan n'est pas un Lepan.");
      }
189

191   Lepan thirdLepan = Lepan.valueOf("NEUTRINO");
      log("Charge de Neutrino : " + thirdLepan.getCharge() );
193 }

195 private static void testMutableEnum(){
      Parfum.VANILLE.setCalories(75);
197   log("Calories dans Vanille: " + Parfum.VANILLE.getCalories());
    }
199
    private static void testEnumRange(){
201   for (Direction direction : EnumSet.range(Direction.NORD, Direction.SUD)){
        log("NORD-SUD: " + direction);
203   }
```

```
  }

  private static void testBitFlags(){
    EnumSet<Direction> directions = EnumSet.of(Direction.EST, Direction.NORD);
    for(Direction direction : directions) {
      log(direction);
    }
  }




  private static void testEnumToStringEtTestValueOf(){
    Direction dir = Direction.valueOf("EST");
    log("Direction toString : " + dir);

    dir = Direction.valueOf("EST");

  }
}
```

```
239
241
243
245
247
249
251
```

- **Exécution :**

```
Exercice d'énumerations...
toString: BOTTOM
UP == UP
UP.equals(UP)
UP avant DOWN
Valeurs de Quark : [LEnumExamples$Quark;@10385c1
Item dans Quark.values() : UP
Item dans Quark.values() : DOWN
Item dans Quark.values() : CHARM
Item dans Quark.values() : STRANGE
Item dans Quark.values() : BOTTOM
Item dans Quark.values() : TOP
toString : Direction: NORD
Est n'est pas froid.
Electron charge : -1
Masse de Lepan : 1.0E-31
Proan n'est pas un Lepan.
Charge de Neutrino : 0
Calories dans Vanille: 75
NORD-SUD: Direction: NORD
NORD-SUD: Direction: SUD
Direction: NORD
Direction: EST
```

0.35.1 Enum et Génériques, EnumSet<...>

```
1
3  import java.util.*;
5  public class test_enum_set {
7    enum Langue {ANGLAIS, FRANCAIS, URDU, JAPONNAIS}
9
11   public static void main(String... aArgs){
       EnumSet<Langue> ariane = EnumSet.of(Langue.FRANCAIS, Langue.ANGLAIS);
13     EnumSet<Langue> noriaki = EnumSet.of(Langue.JAPONNAIS, Langue.ANGLAIS);
15     EnumSet<Langue> tous = EnumSet.allOf(Langue.class);
17     for(Langue lang : tous){
           log(lang);
19     }
       log( "Langues en commun: " + languesCommunesPour(ariane, noriaki) );
21   }
23   private static Set<Langue> languesCommunesPour(Set<Langue> unEnsemble, Set<Langue> →
       ↪ unAutreEnsemble){
       Set<Langue> resultat = new LinkedHashSet<Langue>();
25     for(Langue lang : unEnsemble){
         if( unAutreEnsemble.contains(lang) ) {
27         resultat.add(lang);
         }
29     }
       return resultat;
```

```
31    }
33    private static void log(Object unMassage){
        System.out.println(String.valueOf(unMassage));
35    }
}
37
39
41
43
```

- **Exécution :**

```
ANGLAIS
FRANCAIS
URDU
JAPONNAIS
Langues en commun: [ANGLAIS]
```

0.35.2 Un autre exemple de set d'Enum et ses fonctions

```
1
import java.util.*;
3
public class EnumTest_varargs {
5
enum jusqueQuatre { UN, DEUX, TROIS, QUATRE};
7
public EnumTest_varargs(EnumSet<jusqueQuatre> exUnAQuatre) {
9     affichejusqueQuatres(exUnAQuatre);
      }
11
public EnumTest_varargs(jusqueQuatre... exUnAQuatre) {
13    List<jusqueQuatre> barList = Arrays.asList(exUnAQuatre);
      EnumSet<jusqueQuatre> barSet = EnumSet.copyOf(barList);
15    affichejusqueQuatres(barSet);
      }
17
private static void affichejusqueQuatres(EnumSet<jusqueQuatre> exUnAQuatre) {
19    for (jusqueQuatre b : exUnAQuatre) System.out.print(b.name() + " ");
      System.out.println();
21    }

23 public static void main(String[] args) {
      EnumTest_varargs test1 = new EnumTest_varargs(
25                    EnumSet.complementOf(EnumSet.of(jusqueQuatre.TROIS)));
      jusqueQuatre[] exUnAQuatre = { jusqueQuatre.UN, jusqueQuatre.DEUX, jusqueQuatre.QUATRE };
27    EnumTest_varargs test2 = new EnumTest_varargs(exUnAQuatre);

29    EnumTest_varargs test3 = new EnumTest_varargs(
                      EnumSet.allOf(jusqueQuatre.class));
31    }
```

```
}
33

35

37
```

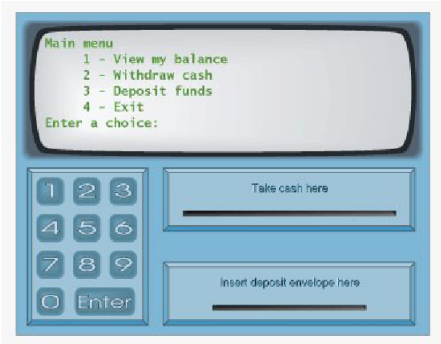
- **Exécution :**

```
 UN DEUX QUATRE
UN DEUX QUATRE
UN DEUX TROIS QUATRE
```

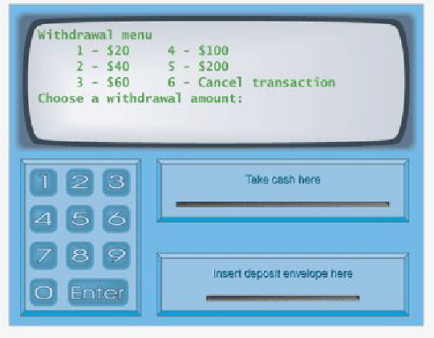
0.36 Exercice GAB

- Ecrire une application Java permettant de gérer un guichet (GAB).

Ecran d'accueil



Menu principal



Ecran de retrait

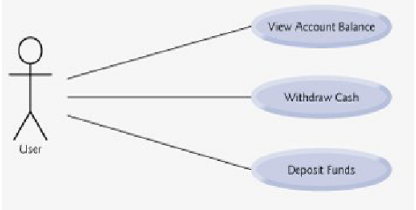


Diagramme Use-Case

0.36.1 UML et GAB

Figure 3.21. Class diagram showing an association among classes.

[View full size image]

FIG. 2 – Class diagram showing an association among classes.

La multiplicité dans une relation en UML :

0	Aucun
1	un
m	une valeur entière
0..1	Zéro ou un
m, n	m ou n
m..n	Au moins m, pas plus de n
*	tout entier non négatif (≥ 0)
0..*	Zéro ou plus (idem *)
1..*	un ou plus

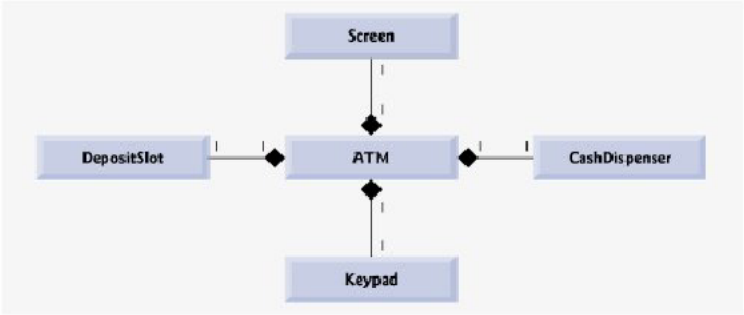


FIG. 3 – Class diagram showing composition relationships.

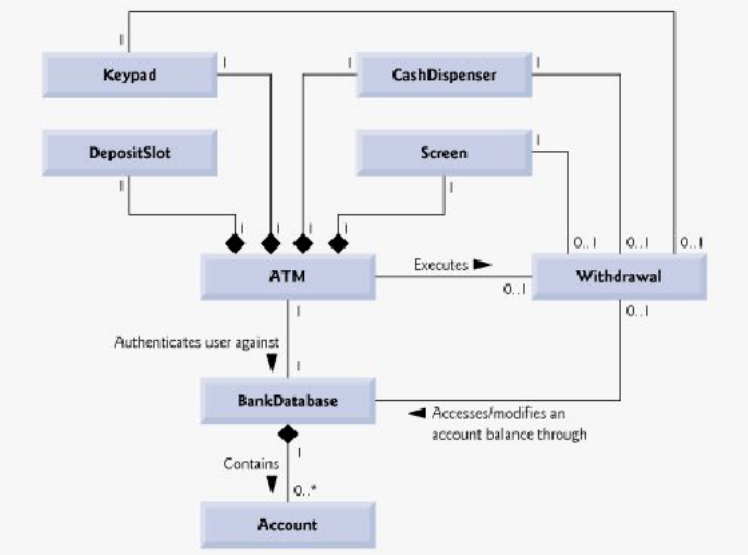


FIG. 4 – . Class diagram for the ATM system model.

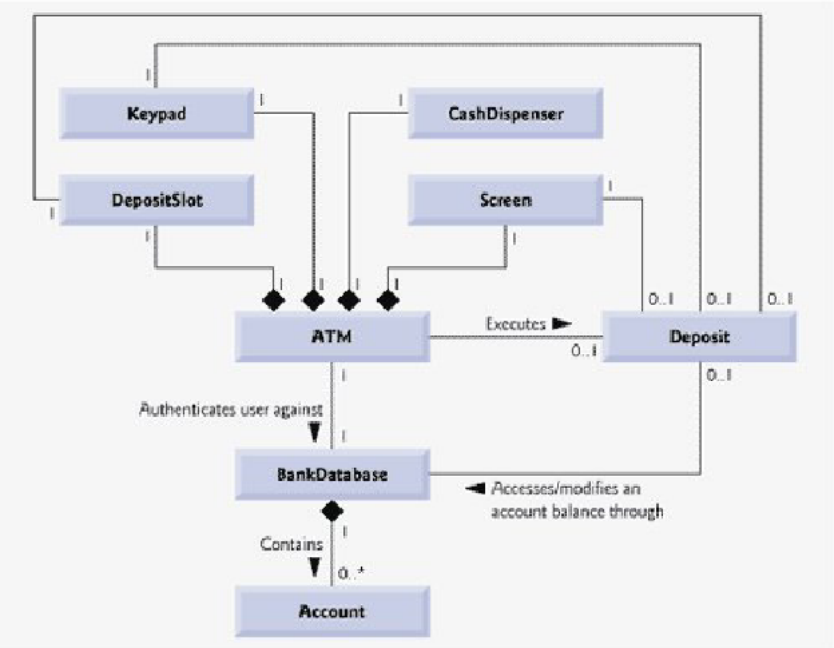


FIG. 5 – Class diagram for the ATM system model including class Dépot.

Arret : page 242 (voir le sommaire)

- Voir pages 319 et 320 pour 2 diag d'activités de GAB (Organigramme)

FIG. 6 – Diagramme de classespour GAB avec attributs.

0.37 Ex Graphique

Prendre l'exemple de conversion celcius Far

0.38 Ex Internationalisation

Prendre l'exemple de Internationalisation d'interface

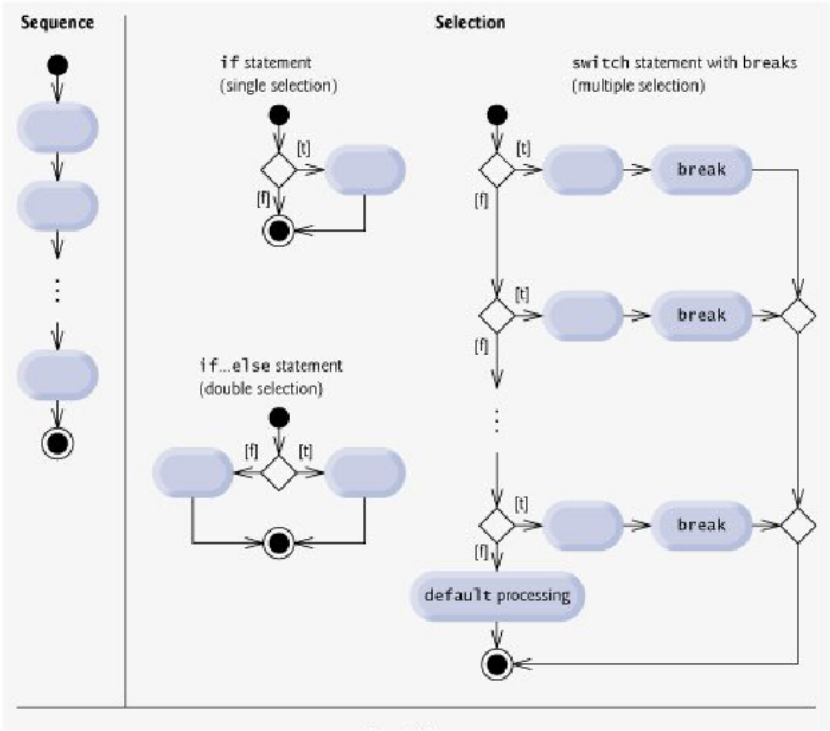


FIG. 7 – Diag activités UML 1

0.39 Ex Editor

Prendre l'exemple Editor.java

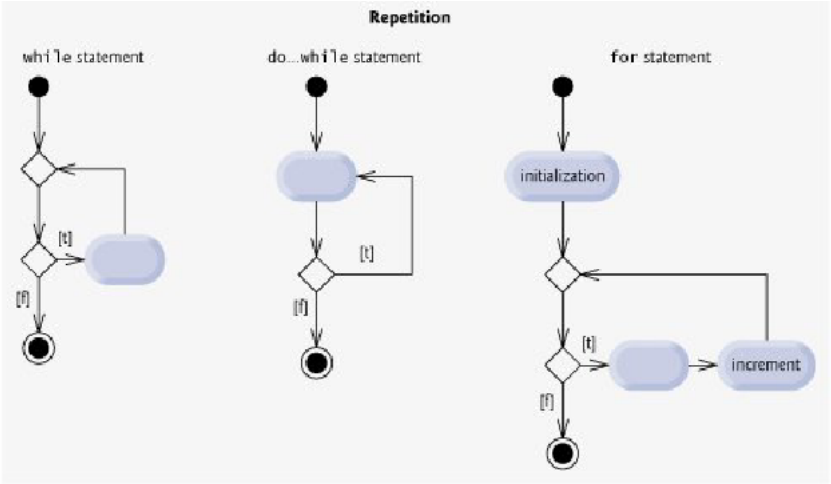


FIG. 8 – Diag activités UML 2

0.40 Ex Applet

Placer Ex_tableau_en_Applet.html

FIG. 9 – Diagramme de classespour GAB avec attributs et méthodes

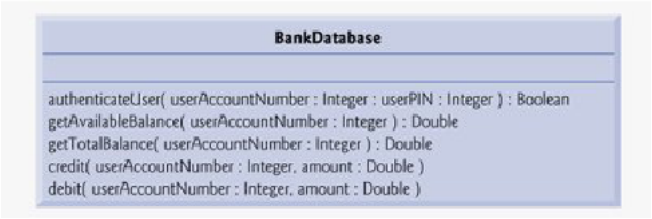


FIG. 10 – Classe BD de la banque

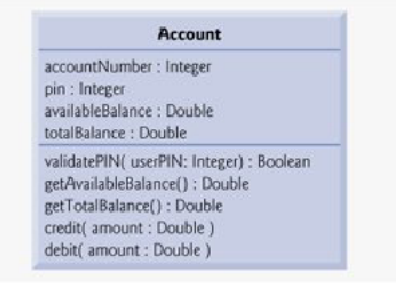


FIG. 11 – Classe Compte de la banque

FIG. 12 – Classe Cash Dispatcher

```
1 begin
```

Listing 6 – rrr

.1 Tables

```
10  z := 0
    for x := 1 to MAX do
12      begin
            write 'Hello World !'
14          z := z + 1
        end
16  write z
```

Et maintenant la suite du programme :

```
    if z > 10 then
18      begin
            write 'Dépassement'
20      end
```

```
for i:=maxint to 0 do
begin
    { do nothing }
end;
```

```
typedef struct {
    Atom_T          *V_ptr;
    Atom_T          *x_ptr;
} ABV_Pair_T;
```

```
1 // 2008 : lecture par scanner (OK)
  // Menu (oui/nom) + lecture d'un string
3 //-----------------------------------------------------------------------
   // import java.util.Scanner;
5    import java.util.*;
  //   import java.io.BufferedReader;
7 //     import java.io.FileReader;
  //     import java.io.IOException;
9 //     import java.io.File;


11


13 public class lect_kb {
```

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

17      java.util.Scanner in = new java.util.Scanner(System.in);
        String reponse = "oui";
19      // OU : String reponse= new String("oui"); /
        String nom= new String ();
21
        // Comparer des adr ? : while (reponse == "oui")
23      while(reponse.equals("oui"))
        {
25
        System.out.println("quel est ton nom?");
```

```
27          nom= in.nextLine();
            System.out.println("salut "+nom+"\n Veux tu recommencer? ( →
              ↪ oui/non)");
29          reponse= in.nextLine();
            }
31
  }
33 }


35


37 //ps: String reponse = new String("oui");
  // peut très bien être écrit comme ca. String reponse = "oui";
```

```
for i:=maxint to 0 do
begin
      { do nothing }
end;
```

'Caption' without label

And we continue the listing:

```
Write('Case insensitive ');
WritE('Pascal keywords.');
```

```
/* normal comment */ % n'apparaît pas
/*+    keep cool   */
/*-     danger!    */
```

Données : D l'ensemble d'apprentissage
Résultat : Un ensemble de k clusters
début
  Choisir les moyennes $m_1, m_2, ..., m_k$ *(e.g. : distance Euclidienne);*
  répéter
    Attribuer tout objet de S *à la moyenne la plus proche;*

    Soient $C_1, C_2, ..., C_k$ *les ensembles d'objets attribués respectivement à* $m_1, m_2, ..., m_k$ *;*
    Ajuster les moyennes par : ;
      $m1 \leftarrow$ *la moyenne de* C_1 *;*
  jusqu'à m_i *ne changent pas (stabilisation des clusters) ;*
  pour tous les i *de* 1 *à* M : $nbr.d'itérations$ **faire**
    % M *: nombre d'itérations fixé en entrée;*
    Découper C *en deux clusters en utilisant Kmeans basique (et minimiser SSE);*
  fin
  pour tous les *Core points* **faire**
    si *Le Core point n'a pas de label de cluster* **alors**
      Lable_cluster_actuel := Lable_cluster_actuel +1;
      Donner le label Lable_cluster_actuel au Core point courant;
    fin
  fin
fin

Algorithme 1 : K-Means basique

Listings

Table des matières

Liste des Algorithmes