# Programmation objet et JAVA





# 0

## Bibliographie


**Livre de référence**

- Gilles Roussel, Étienne Duris, Nicolas Bedon, Rémi Forax, *Java et Internet : Concepts et Programmation, Tome 1 : côté client*, 2ème édition , Vuibert, novembre 2002.

Notes de cours et transparents d'Étienne Duris, Rémi Forax, Dominique Perrin, Gilles Roussel.

**Autres ouvrages sur Java**

- Cay S. Horstmann, Gary Cornell, *Au coeur de Java 2*, Sun Microsystem Press (Java Series).
  - Volume I - Notions fondamentales, 1999.
  - Volume II - Fonctions avancées, 2000.
- Ken Arnold, James Gosling, *The Java Programming Language Second edition*, Addison Wesley, 1998.
- Samuel N. Kamin, M. Dennis Mickunas, Edward M. Reingold, *An Introduction to Computer Science Using Java*, McGraw-Hill, 1998.





- Patrick Niemeyer, Joshua Peck (Traduction de Eric Dumas), *Java par la Pratique*, O'Reilly International Thomson, 1996.
- Matthew Robinson and Pavel Vorobiev, *Swing*, Manning Publications Co., december 1999.
  (voir `http://manning.spindoczine.com/sbe/`)

**Sur les Design Pattern** Le livre de référence est

- Erich Gamma, Richard Helm, Ralph Johnsons, John Vlissides, *Design Patterns*, Addison-Wesley, 1995. Traduction française chez Vuibert, 1999.


Souvent désigné par GoF (Gang of Four).



# 1

## Programmation objet

1. Styles de programmation
2. Avantages du style objet
3. Premiers exemples

## Styles de programmation

***Style applicatif***

- Fondé sur l'évaluation d'expressions, où le résultat ne dépend que de la valeurs des arguments (et non de l'état de la mémoire).
- Donne programmes courts, faciles à comprendre.
- Usage intensif de la récursivité.
- Langage typique : Lisp, Caml.

***Style impératif***

- Fondé sur l'exécution d'instructions modifiant l'état de la mémoire.
- Utilise une structure de contrôle et des structures de données.
- Usage intensif de l'itération.
- Langages typiques : Fortran, C, Pascal.

***Style objet***

- Un programme est vu comme une communauté de composants autonomes (objets) disposant de ses ressources et de ses moyens d'interaction.
- Utilise des classes pour décrire les structures et leur comportement.
- Usage intensif de l'échange de message (métaphore).
- Langages typiques : Simula, Smalltalk, C++, Java, Ocaml.



## Avantages du style objet

***Facilite la programmation modulaire***

- La conception par classes conduit à des composants réutilisables.
- Un composant offre des services, et en utilise d'autres.
- Il "expose" ses services à travers une *interface*.
- Il cache les détails d'implémentations (encapsulation ou data-hiding).
- Ceci le rend réutilisable.

***Facilite l'abstraction***

- L'abstraction sépare la définition de son implémentation.
- L'abstraction extrait un modèle commun à plusieurs composants.
- Le modèle commun est partagé par le mécanisme d'*héritage*.

***Facilite la spécialisation***

- La spécialisation traite de cas particuliers.
- Le mécanisme de dérivation rend les cas particuliers transparents.



## Premier exemple

Le fichier
`HelloWorld.java` :

```
class HelloWorld {
  public static void main (String[] args) {
    System.out.println("Bonjour à tous !");
  }
}
```

Compilation :
`javac HelloWorld.java`

crée le fichier `HelloWorld.class`

Exécution :
`java HelloWorld`

Résultat :
`Bonjour à tous !`

- Il est usuel de donner une initiale majuscule aux classes, et une initiale minuscule aux attributs et aux méthodes.
- Le nom du fichier qui contient le code source est en général le nom de la classe suffixé par `.java`.



## Le langage

Java (nom dérivé de Kawa) a vu le jour en 1995. Actuellement version JDK 1.6 (aussi appelée Java 6). Environnements d'exécution : J2ME, J2SE, J2EE.

Java

- est fortement typé,
- est orienté objet,
- est compilé–interprété,
- intègre des *thread* ou processus légers,
- est sans héritage multiple,
- à partir de la version 1.5, Java offre de la généricité (les `generics` sont différents des *template* du C++)

Compilation – Interprétation

- Source *compilée* en langage intermédiaire (*byte code*) indépendant de la machine cible.
- Byte code *interprété* par une *machine virtuelle Java* (dépendant de la plateforme).

Avantages : l'exécution peut se faire

- plus tard,
- ailleurs (par téléchargement).

Des milliers de classes prédéfinies qui encapsulent des mécanismes de base :

- Structures de données : vecteurs, listes, ensembles ordonnés, arbres, tables de hachage, grands nombres;
- Outils de communication, comme les URL, client-serveur;
- Facilités audiovisuelles, pour images et son;
- Des composants de création d'interfaces graphiques;
- Traitement de fichiers;
- Accès à des bases de données.

Variables d'environnements :

- JAVA_HOME correspond au répertoire racine du JDK.
- CLASSPATH correspond aus répertoires contenant des classes du développeur.

Voir aussi les options du compilateurs.



Exemple 2

```
class Hello {
  public static void main (String[] args) {
    String s = "Hello ";
    s = s + args[0]; // contanénation des chaînes
    System.out.println(s);
  }
}
```

Exécution :
`java Hello David`

Résultat :
`Hello David`
ou encore

```
class Hello2 {
  public static void main (String[] args) {
      StringBuilder sb = new StringBuilder();
      sb.append("Hello ");
      sb.append(args[0]);
      System.out.println(sb);
  }
}
```

Même résultat. Les classes `StringBuilder et StringBuffer` permettent de manipuler les chaînes de façon plus efficaces. La classe `StringBuffer` est sécurisée pour les threads.

Les entrées sorties sont facilitées avec la classe `java.util.Scanner`.



```
import java.util.*;

class Hello3 {
  public static void main (String[] args) {
      String s;
      Scanner sc = new Scanner(System.in);
      while (sc.hasNext()){
          s = sc.next();
          System.out.println(s);
      }
      sc.close();
  }
}
```

On peut lire un entier facilement.

```
import java.util.*;

class MyRead {
  public static void main (String[] args) {
      Scanner sc = new Scanner(System.in);
      int i = sc.nextInt();
      System.out.println(i);
      sc.close();
  }
}
```



Exemple 3

```
import java.awt.*;
/**
 * Classe affichant une fenetre de nom "Hello David"
 * contenant "Hello David".
 * @author Beal
 * @version 1.0
 */
public class HelloWorldCanvas extends Canvas{
  /**
   * Constructeur
   */
  public HelloWorldCanvas() {
    super();
  }
  /**
   * Methode de dessin de la fenetre
   * @param graphics contexte d'affichage
   */
  public void paint(Graphics graphics) {
    graphics.setColor(Color.black);
    graphics.drawString("Hello David",65,60);
  }
  /**
   * Methode main
   * @param args arguments de la ligne de commande
   */
  public static void main(String[] args) {
    HelloWorldCanvas c = new HelloWorldCanvas();
    Frame f = new Frame("Hello David");
    f.setSize(200,200);
    f.add(c);
    f.setVisible(true);
  }
}
```

## Exemple des points

```java
public class Pixel {
  private int x;
  private int y;

  public Pixel (int x, int y) {
    this.x = x; this.y = y;
  }
  public int getX() {
    return x;
  }
  public int getY() {
    return y;
  }
  public void setX(int x) {
    this.x = x;
  }
  public void setY(int y) {
    this.y = y;
  }
  public void move(int dx, int dy) {
    x += dx; y += dy;
  }
  @Override
  public String toString() {
    return(this.x + ", " + this.y);
  }
  public static void main(String[] args) {
    Pixel a = new Pixel(3,5);
    a.setY(6);      // a = (3,6)
    a.move(1,1);    // a = (4,7)
    System.out.println(a);
  }
}
```



## Héritage et composition

***Par héritage, une classe dérivée bénéficie des attributs et des méthodes de la superclasse.***

- La classe dérivée possède les attributs et les méthodes de la classe de base.
- La classe dérivée peut en ajouter, ou en masquer.
- Facilite la programmation par raffinement.
- Facilite la prise en compte de la spécialisation.

***Par la composition, une classe utilise un autre service.***

- Un composant est souvent un attribut de la classe utilisatrice.
- L'exécution de certaines tâches est *délégué* au composant le plus apte.
- Le composant a la *responsabilité* de la bonne exécution.
- Facilite la séparation des tâches en modules spécialisés.



## Exemple : disques et anneaux

```java
public class Disk {
  protected Pixel center;  // composition
  protected int radius;
  public Disk(int x, int y, int radius) {
    center = new Pixel(x,y);
    this.radius = radius;
  }
  @Override
  public String toString() {
    return center.toString() + " ," + radius;
  }
  public void move(int dx, int dy) {
    center.move(dx, dy);  // délégation
  }
}

public class Ring extends Disk { // dérivation
  private int internalRadius;
  public Ring(int x, int y, int radius, int internalRadius) {
    super(x, y, radius);
    this.internalRadius = internalRadius;
  }
  @Override
  public String toString() {
    return super.toString() + " ," + internalRadius;
  }
}
class Test {
  public static void main(String[] args) {
    Ring a = new Ring(3, 5, 7, 2);
    a.move(5,-2); // héritée de Disk, qui délègue à Pixel
    System.out.println(a); // 8, 3 , 7, 2
  }
}
```



# 2

## Le langage Java 1.6

## Structure d'un programme

```
class HelloWorld {
  public static void main (String[] args) {
    System.out.println("Hello World !");
  }
}
```

*Programme Java* : constitué d'un ensemble de classes

- groupées en paquetages (*packages*);
- réparties en fichiers;
- chaque classe compilée est dans son propre fichier (un fichier dont le nom est le nom de la classe suffixé par `.class`).

*Un fichier source Java* comporte

- des directives d'importation comme

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

- des déclarations de classes.



*Une classe* est composée de *membres* :

- déclarations de variables (*attributs*);
- définitions de fonctions (*méthodes*);
- déclarations d'autres classes (*nested classes*);

*Les membres sont*

- des membres de classe (`static`);
- des membres d'objet (ou d'instance).

Une classe a trois rôles:

1. de typage, en déclarant de nouveaux types;
2. d'implémentation, en définissant la structure et le comportement d'objet;
3. de moule pour la création de leurs instances.

*Une méthode* se compose

- de déclarations de variables locales;
- d'instructions.

Les types des paramètres et le type de retour constituent la *signature* de la méthode.

```
static int pgcd(int a, int b) {
  return (b == 0) ? a : pgcd( b, a % b );
}
```

*Point d'entrée* : Une fonction spéciale est appelée à l'exécution. Elle s'appelle toujours `main` et a toujours la même signature.

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



***Toute méthode, toute donnée fait partie d'une classe*** (pas de variables globales). L'appel se fait par déréférencement d'une classe ou d'un objet d'une classe, de la façon suivante :

- Méthodes ou données *de classe* : par le nom de la classe.

  ```
  Math.cos()      Math.PI
  ```

- Méthode ou donnée d'un objet : par le nom de l'objet.

  ```
  ...
  Stack s = new Stack();
  s.push(x);
  ```

- L'objet courant est nommé `this` et peut être sous-entendu.

  ```
  public void setX(int x) {
    this.x = x;
  }
  ```

- La classe courante peut être sous-entendue pour des méthodes statiques.

Exemple :

```
System.out.println()
```

- `out` est un membre statique de la classe `System`.
- `out` est un objet de la classe `PrintStream`.
- `println` est une méthode d'objet de la classe `PrintStream`.



## Expressions, types, variables

Toute expression a une valeur et un type. Les valeurs sont

- les valeurs primitives;
- les références, qui sont des références à des tableaux ou à des objets.
- Il existe une référence spéciale `null`. Elle peut être la valeur de n'importe quel type non primitif.

***Un objet ne peut être manipulé, en Java, que par une référence.***

Une *variable* est le nom d'un emplacement mémoire qui peut contenir une valeur. Le *type* de la variable décrit la nature des valeurs de la variable.

- Si le type est un type primitif, la valeur est de ce type.
- Si le type est une classe, la valeur est une référence à un objet de cette classe, ou d'une classe dérivée. Une référence est différente des pointeurs du C (pas d'arithmétique dessus).

Exemple :

```
Pixel p;
```

déclare une variable de type `Pixel`, susceptible de contenir une référence à un objet de cette classe.

```
p = new Pixel(4, 6);
```

L'évaluation de l'expression `new Pixel(4, 6)` retourne une référence à un objet de la classe `Pixel`. Cette référence est affectée à `p`.

Passage de paramètres

Toujours par valeur.

Exemple : Soit la méthode

```
static int plus(int a, int b) {
  return a+b;
}
```

À l'appel de la méthode, par exemple `int c = plus(a+1,7)`, les paramètres sont évalués, des variables locales sont initialisées avec les valeurs des paramètres, et les occurrences des paramètres formels sont remplacées par les variables locales correspondantes. Par exemple,

```
int aLocal = a+1;
int bLocal = 7;
résultat = aLocal+bLocal;
```

Attention : ***Les objets sont manipulés par des références. Un passage par valeur d'une référence est donc comme un passage par référence !***

Attention : Ce n'est pas le passage par référence du C++.

Exemple :

```
static void increment(Pixel a) {
  a.x++; a.y++;
}
```

Après appel de `increment(a)`, les coordonnées du point sont incrémentées !



Une variable se déclare en donnant d'abord son type.

```
int i, j = 5;
float re, im;
boolean termine;
static int numero;
static final int N = 12;
```

A noter :

- Une variable peut être initialisée.
- Une variable `static` est un membre de classe.
- Une variable `final` est une constante.
- Tout attribut de classe est initialisé par défaut, à `0` pour les variables numériques, à `false` pour les booléennes, à `null` pour les références.
- Dans une *méthode*, une variable doit être déclarée avant utilisation. Elle n'est pas initialisée par défaut.
- Dans la définition d'une *classe*, un attribut peut être déclaré après son utilisation. (Elle se fait *à l'intérieur* d'une méthode.)



## Instructions

Affectation, instructions conditionnelles, aiguillages, itérations usuelles.

*Affectation :*

```
x = 1; y = x = x+1;
```

*Instructions conditionnelles :*

```
if (C) S
if (C) S else T
```

*Itérations :*

```
while (C) S
do S while (C)
for (E; C; G) S
```

Une instruction `break;` fait sortir du bloc où elle se trouve.

La conditionnelle C doit être de type booléen



*Traitement par cas :*

```
switch(c) {
  case ' ':
    nEspaces++; break;
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9':
    nChiffres++; break;
  default:
    nAutres++;
}
```

*Blocs à étiquettes :*

```
un: while (...) {
  ...
  deux : for (...) {
    ...
    trois: while (...) {
      ...
      if (...) continue un; // reprend while exterieur
      if (...) break deux;  // quitte boucle for
      continue; // reprend while interieur
    }
  }
}
```

## Types primitifs

| Nom | Taille | Exemples |
|---|---|---|
| `byte` | 8 | 1, -128, 127 |
| `short` | 16 | 2, 300 |
| `int` | 32 | 234569876 |
| `long` | 64 | 2L |
| `float` | 32 | 3.14, 3.1E12, 2e12 |
| `double` | 64 | 0.5d |
| `boolean` | 1 | `true` ou `false` |
| `char` | 16 | `'a'`, `''\n'`, `'\u0000'` |

A noter :

- Les caractères sont codés sur *deux octets* en Unicode.
- Les types sont *indépendants* du compilateur et de la plate-forme.
- Tous les types numériques sont signés sauf les caractères.
- Un booléen n'est pas un nombre.
- Les opérations sur les entiers se font modulo, et sans erreur :

```
byte b = 127;
b += 1; // b = -128
```



## Enveloppes des types primitifs

| | |
|---|---|
| `boolean` | java.lang.Boolean |
| `char` | java.lang.Character |
| `byte` | java.lang.Byte |
| `short` | java.lang.Short |
| `int` | java.lang.Integer |
| `long` | java.lang.Long |
| `float` | java.lang.Float |
| `double` | java.lang.Double |

- Une instance de la classe enveloppe encapsule une valeur du type de base correspondant.
- Chaque classe enveloppe possède des méthodes pour extraire la valeur d'un objet : `primitifValue()` appliquée sur l'objet enveloppe renvoie une valeur de type `primitif`.
- Une méthode statique de chaque classe `Enveloppe` : `Enveloppe.valueOf(primitif p)` renvoie un objet enveloppant le primitif correspondant.
- Un objet enveloppant est immutable : la valeur contenue ne peut être modifiée.
- On transforme souvent une valeur en objet pour utiliser une méthode manipulant ces objets.



## Conversions automatiques (auto-boxing et auto-unboxing)

Depuis la version 1.5, la conversion est automatique.
*Auto-boxing*

```
Integer i = 3; // int -> Integer
Long l = 3L;  // long -> Long
Long l = 3; // erreur, int -> Integer -X-> Long
```

*Auto-unboxing*

```
Integer i = new Integer(3);
int x = i; // Integer -> int
Long lo = null;
long l = lo; //erreur : java.lang.NullPointerException
```

*Auto-boxing et appels de méthodes*

```
class Test3{
  static void myPrint(Integer i){
    System.out.println(i);
  }
  public static void main(String[] args){
     myPrint(5); //affiche 5
  }
}
```

*Auto-boxing et égalité*
Sur des objets, `==` teste l'égalité des références. Les enveloppes obtenues par auto-boxing ont la même référence.



```
public static void main(String[] args){
        Long a = 5L;
        Long b = 5L;
        Integer i = 6;
        Integer j = 6;
        Integer k = new Integer(6);
        System.out.println(a == b); //true, a,b petits
        System.out.println(i == j); //true, i,j petits
        System.out.println(i == k); //false
        i = i+1;//Integer -> int +1 -> int -> Integer
        System.out.println(i); // 7
        System.out.println(j); // 6
    }
```

Ne pas tester l'égalité de références.

## Sous-typage

Le sous-typage est la possibilité de référencer un objet d'un certain type avec un autre type.

- En Java, le sous-typage coïncide avec la dérivation.
- Il n'y a pas de relation de sous-typage entre types primitifs et types objets.
- Le conversions d'auto-boxing et auto-unboxing sont faites avant les transtypages sur les types objets.

Par exemple, `Integer`, `Long`, `Float`, `Double` dérivent de la classe abstraite (voir plus loin dans le cours) `Number`.

```
Number n = new Integer(3);
Number m = new Double(3.14);
Object o =  new Integer(3);
Object obis = m;
Integer i = new Object();//erreur
```



## Tableaux

C'est un objet particulier. L'accès se fait par référence et la création par `new`. Un tableau

- se *déclare*,
- se *construit*,
- et s'*utilise*.

*Identificateur* de type tableau se déclare par

```
int[] tab; // vecteur d'entiers
double[][] m; // matrice de doubles
```

→ La déclaration des tableaux comme en C est acceptée mais celle-ci est meilleure.

La valeur de l'identificateur n'est pas définie après la déclaration.

*Construction* d'un tableau par `new` :

```
tab = new int[n] ;
m = new double[n][p] // n lignes, p colonnes
```

*Utilisation* traditionnelle

```
int i, j;
m[i][j] = x;  // ligne i, colonne j
for (i = 0; i < tab.length; i++)
  System.out.print(tab[i]);
//ou boucle for each
for (int x:tab) System.out.print(x);
```



Tout tableau a un attribut `length` qui donne sa taille à la création. Distinguer:

- la déclaration, qui concerne la variable dont le contenu sera une référence sur un tableau,
- la construction, qui crée le tableau et retourne une référence sur ce tableau.

On peut fusionner déclaration et construction par initialisation énumérative :

```
String[] jours = {"Lundi", "Mardi", "Mercredi",
  "Jeudi", "Vendredi", "Samedi", "Dimanche" };
```

Les instructions suivantes provoquent toujours une exception (de la classe `ArrayIndexOutOfBoundsException`) :

`a[a.length]`,

`a[-1]`.



*Un exemple* :

```
/**
 * Matrices of integers.
 * @author Beal
 * @author Berstel
 * @version 1.0
 */

public class Matrix {
  int[][] m;

   /**
   * Create a null matrix.
   * @param dim dimension of the matrix
   * @see Matrix#Matrix(int,int)
   */

   public Matrix(int dim) {
      this(dim,0);
  }

     /**
   * Create a matrix whose coefficients are equal to a same number
   * @param dim dimension of the matrix
   * @param x integer value of each coefficient
   * @see Matrix#Matrix(int)
   */

   public Matrix(int dim, int n) {
      m = new int [dim][dim];
      for (int i = 0; i < dim; i++)
       for (int j = 0; j < dim; j++)
          m[i][j] = n;
  }

 /**
   * Transpose this matrix
   */

 public void transposer() {
```

```
  int dim = m.length;
  for (int i = 0; i < dim; i++)
    for (int j = i+1; j < dim; j++){
    int t = m[i][j]; m[i][j] = m[j][i]; m[j][i]=t;
    }
}

   /**
   * Returns a String object representing this Matrix's value.
   */


   @Override
   public String toString(){
     StringBuilder sb = new StringBuilder();
     for (int[] t : m){
          for (int x : t){
             sb.append(x);
             sb.append(" ");
          }
          sb.append("\n");
     }
     return sb.toString();
   }

   /**
  * Main
  * @param args arguments of the line command
  */

  public static void main(String[] args) {
    Matrix a = new Matrix(3,12);
    System.out.print(a);
    Matrix b = new Matrix(3,3);
    System.out.print(b);
    Matrix c = new Matrix(3);
    System.out.print(c);
  }
}
```



```
$java Matrix
12 12 12
12 12 12
12 12 12
3 3 3
3 3 3
3 3 3
0 0 0
0 0 0
0 0 0
```

Création de la documentation HTML

```
javadoc Matrix.java
```

Visualisation avec Netscape.

```
$ ls
allclasses-frame.html    Matrix.java
allclasses-noframe.html  Matrix.java~
constant-values.html     overview-tree.html
deprecated-list.html     package-frame.html
help-doc.html            package-list
index-all.html           package-summary.html
index.html               package-tree.html
Matrix.class             resources/
Matrix.html              stylesheet.css
```



## Créer une documentation HTML

Il s'agit de créer une documentation et non d'établir les spécifications des classes.

- `@author` : il peut y avoir plusieurs auteurs;
- `@see` : pour créer un lien sur une autre documentation Java;
- `@param` : pour indiquer les paramètres d'une méthode;
- `@return` : pour indiquer la valeur de retour d'une méthode;
- `@exception` : pour indiquer quelle exception est levée;
- `@version` : pour donner le numéro de version du code;
- `@since` : pour donner le numéro de la version initiale;
- `@deprecated` : indique une méthode ou membre qui ne devrait plus être utilisé. Crée un warning lors de la compilation.
  - Exemple :

```
/**
* @deprecated Utiliser plutot afficher()
* @see #afficher()
*/
```



## Méthodes

Chaque classe contient une suite non emboîtée de méthodes : on ne peut pas définir des méthodes à l'intérieur de méthodes.

```
static int next(int n) {
  if (n % 2 == 1)
    return 3 * n + 1;
  return n / 2;
}

static int pgcd(int a, int b) {
  return (b == 0) ? a : pgcd( b, a % b );
}
```

Une méthode qui ne retourne pas de valeur a pour type de retour le type `void`.

## Surcharge

On distingue :

- *Profil* : le nom plus la suite des types des arguments.
- *Signature* : le type de retour plus le profil.
- *Signature complète* : signature plus la visibilité (`private`, `protected`, `public`, ou rien).
- *Signature étendue* : signature complète plus les exceptions.

Un même identificateur peut désigner des méthodes différentes pourvu que leurs profils soient différents.

```
static int fact(int n, int p) {
  if (n == 0) return p;
  return fact( n-1, n*p);
}

static int fact(int n) {
  return fact(n, 1);
}
```

Il n'y a pas de valeurs par défaut (comme en C++). Il faut donc autant de définitions qu'il y a de profils.



## Visibilité des attributs et méthodes

Les membres (attributs ou méthodes) d'une classe ont une visibilité définie par défaut et ont des modificateurs de visibilité :

```
public
protected
private
```

Par défaut, une classe a ses données ou méthodes accessibles dans le répertoire, plus précisément dans le paquetage dont il sera question plus loin.
Un attribut (données ou méthodes)

- `public` est accessible dans tout code où la classe est accessible.
- `protected` est accessible dans le code des classes du même paquetage et dans les classes dérivées de la classe.
- `private` n'est accessible que dans le code de la classe.

La méthode `main()` doit être accessible de la machine virtuelle, donc doit être `public`.



## Constructeurs

Les objets sont instanciés au moyen de constructeurs. Toute classe a un constructeur par défaut, sans argument. Lors de la construction d'un objet, l'opérateur `new` réserve la place pour l'objet et initialise les attributs à leur valeur par défaut.
Le constructeur

- exécute le corps de la méthode;
- retourne la référence de l'objet créé.

Exemple avec seulement le constructeur par défaut :

```
class Pixel {
  int x,y;
}
```

Utilisation :

```
public static void main(String[] args) {
  Pixel p;         // p est indéfini
  p = new Pixel(); // p!= null, p.x = p.y = 0;
  p.x = 4;
  p.y = 5;
  ...
}
```



Exemple avec un constructeur particulier :

```
class Pixel {
  int x,y;
  Pixel(int x, int y) {
    this.x = x; this.y = y;
  }
}
public static void main(String[] args) {
  Pixel p, q;         // p, q indéfinis
  p = new Pixel(2,3); // p.x = 2, p.y = 3;
  q = new Pixel();    // erreur !
  ...
}
```

La définition explicite d'un constructeur fait disparaître le constructeur par défaut implicite. Si l'on veut garder le constructeur défini et le constructeur par défaut, il faut alors déclarer explicitement celui-ci.

```
class Pixel {
  int x,y;
  Pixel() {}
  Pixel(int x, int y) {
    this.x = x; this.y = y;
  }
}
public static void main(String[] args) {
  Pixel p, q;         // p,q indéfinis
  p = new Pixel(2,3); // p.x = 2, p.y = 3;
  q = new Pixel();    // OK !
  ...
}
```

Les données d'un objet peuvent être des (références d') objets.

```
class Segment {
  Pixel start ;
  Pixel end;
}
```

Utilisation :

```
public static void main(String[] args) {
  Segment s;   // s indéfini
  s = new Segment() ; // s.start = null, s.end = null
}
```

Plusieurs constructeurs pour la même classe :

```
class Segment {
  Pixel start;
  Pixel end;
  Segment() {} // par defaut
  Segment(Pixel start, Pixel end) {
    this.start = start;
    this.end = end;
  }
  Segment(int dx, int dy, int fx, int fy) {
    start = new Pixel(dx, dy);
    end = new Pixel(fx, fy);
  }
}
```



Noter que

- dans le deuxième constructeur, on affecte à `start` et `end` les références d'objets existants;
- dans le troisième, on crée des objets à partir de données de base, et on affecte leurs références.

Exemples d'emploi :

```
public static void main(String[] args) {
  Segment s;   // s indéfini
  s = new Segment() ; // s.start = null, s.end = null
  s.start = new Pixel(2,3);
  s.end = new Pixel(5,8);
  Pixel p = new Pixel(2,3);
  Pixel q = new Pixel(5,8);
  Segment t = new Segment(p,q);
  Segment tt=
    new Segment(new Pixel(2,3), new Pixel(5,8));
  Segment r = new Segment(2,3,5,8);
}
```



## Membres et méthodes statiques

Les attributs peuvent être

- des attributs de classe (`static`);
- des attributs d'objet (ou d'instance).

Les attributs de classe `static` sont partagés par tous les objets de la classe. Il n'en existe qu'un par classe au lieu de un pour chaque instance ou objet d'une classe lorsqu'il s'agit de membre d'objets.

*Exemple d'attributs* `static`*:*

- un compteur du nombre d'objets;
- un élément particulier de la classe, par exemple une origine.

```
class Pixel {
   int x, y;
   static Pixel origin = new Pixel(0,0);
}
```

Les méthodes peuvent aussi être ou non `static`.

Les méthodes statiques sont appelées en donnant le nom de la classe ou le nom d'une instance de la classe. Une méthode statique ne peut pas faire référence à `this`.

Elles sont utiles pour fournir des services (helper). Méthodes de la classe `Math`.



## Exemple 1

```
public class Chrono {
  private static long start, stop;
  public static void start() {
    start = System.currentTimeMillis();
  }
  public static void stop() {
    stop = System.currentTimeMillis();
  }
  public static long getElapsedTime() {
        return stop - start;
  }
}
```

On s'en sert comme dans

```
class Test {
  public static void main(String[] args) {
    Chrono.start();
      for (int i = 0; i < 10000; i++)
        for (int j = 0; j < 10000; j++);
    Chrono.stop();
    System.out.println("Duree = "+ Chrono.getElapsedTime());
  }
}
```

Sur cet exemple, on peut aussi créer un objet chronomètre avec des méthodes dynamiques.

**Exemple 2**

```
class User {
  String nom;
  static int nbUsers;
  static User[] allUsers = new User[10];

  User(String nom) {
          this.nom = nom;
    allUsers[nbUsers++] = this;
  }
  void send(String message, User destinataire) {
    destinataire.handleMessage( message, this);
  }
  void handleMessage(String message, User expediteur) {
        System.out.println(
      expediteur.nom + " dit \"" + message + "\" à " + nom);
  }
  void sendAll(String message) {
    for (int i = 0; i < nbUsers; i++)
      if (allUsers[i] != this) send( message, allUsers[i]);
  }


  public static void main(String[] args) {
    User a = new User("Pierre"), b = new User("Anne"),
      c = new User("Alex"), d = new User("Paul");
    a.send("Bonjour", b);
    b.sendAll("Hello");
    a.sendAll("Encore moi");
  }
}
```



On obtient :

```
Pierre dit "Bonjour" à Anne
Anne dit "Hello" à Pierre
Anne dit "Hello" à Alex
Anne dit "Hello" à Paul
Pierre dit "Encore moi" à Anne
Pierre dit "Encore moi" à Alex
Pierre dit "Encore moi" à Paul
```



## Méthodes avec un nombre variable d'arguments : `varargs`

Depuis Java 1.5, il est possible de passer dans une méthode un nombre variable d'arguments de même type.

- Dans le code, les arguments sont traités comme un tableau.
- Lors de l'appel, on peut passer une suite ou un tableau.

```
class TestVarArgs{
static double polygonLength(Pixel... pixels){
 float sum = 0;
 int x = pixels[0].getX();
 int y = pixels[0].getY();
 double squareOfSide, lengthOfSide;
 for (int i=1; i < pixels.length; i++) {
    squareOfSide =
       (pixels[i].getX()-x)*(pixels[i].getX()-x)
     + (pixels[i].getY()-y)*(pixels[i].getY()-y);
    lengthOfSide = Math.sqrt(squareOfSide);
    x = pixels[i].getX();
    y = pixels[i].getY();
    sum += lengthOfSide;
 }
 return sum;
}
```



```
public static void main(String... args) {
  Pixel p1= new Pixel(3,2);
  Pixel p2 = new Pixel(3,4);
  Pixel p3 = new Pixel(3,10);
  System.out.println(polygonLength(p1,p2,p3));
  Pixel[] tab = new Pixel[3];
  tab[0] = p1;  tab[1] = p2;  tab[2] = p3;
  System.out.println(polygonLength(tab));
}
}
```

Quelle est l'interêt de la syntaxe `varargs` ?

## Héritage : généralités

L'*héritage* consiste à faire profiter tacitement une classe *dérivée* $D$ des attributs et des méthodes d'une classe *de base* $B$.

$$\begin{array}{c} B \\ \uparrow \\ D \end{array}$$

- La classe dérivée possède les attributs de la classe de base (et peut y accéder sauf s'ils sont privés).
- La classe dérivée possède les méthodes de la classe de base (même restriction).
- La classe dérivée peut déclarer de nouveaux attributs et définir de nouvelles méthodes.
- La classe dérivée peut redéfinir des méthodes de la classe de base. La méthode redéfinie *masque* la méthode de la classe de base.
- Dérivation par `extends`.
- Toute classe dérive, directement ou indirectement, de la classe `Object`.
- L'arbre des dérivations est visible dans les fichiers créés par `javadoc`, sous eclipse,...
- La relation d'héritage est transitive.



```
class Base {
  private int p = 2;
  int x = 3, y = 5;
  @Override public String toString(){
    return "B "+p+" "+x+" "+y;
  }
  int sum(){return x+y;}
}
class Der extends Base {
  int z=7;
  @Override public String toString(){
    return "D "+x+" "+y+" "+z;
  }
}
public class BaseTest{
  public static void main(String[] args) {
    Base b = new Base();
    Der d = new Der();
    System.out.println(b);
    System.out.println(b.sum());
    System.out.println(d);
    System.out.println(d.sum());
  }
}
```

Le résultat est :

```
B 2 3 5
8
D 3 5 7
8
```



## Usages

Une classe dérivée représente

- une *spécialisation* de la classe de base.
  Mammifère dérive de vertébré, matrice symétrique dérive de matrice. Compte sur livret A dérive de Compte bancquaire.
- un *enrichissement* de la classe de base.
  Un segment coloré dérive d'un segment. Un article a un prix, un vêtement est un article qui a une taille. Un aliment est un article qui a une date de péremption.

Une classe de base représente des propriétés communes à plusieurs classes. Souvent c'est une *classe abstraite*, c'est-à-dire sans réalité propre.

- Une *figure* est une abstraction d'un rectangle et d'une ellipse.
- Un *sommet* est un nœud interne ou une feuille.
- Un *vertébré* est une abstraction des *mammifères* etc. Les mammifères eux-mêmes sont une abstraction.
- Un *type abstrait de données* est une abstraction d'une *structure de données*.

La hauteur des dérivations n'est en général pas très élévée. Il ne faut pas "se forcer" à créer des classes dérivées.



## Points épais : exemple de dérivation

Il s'agit d'un point auquel on ajoute une information, l'épaisseur. On a donc un enrichissement.

```
class Pixel {
  int x, y;
  Pixel (int x, int y) {
    this.x = x; this.y = y;
  }
  void translate(int dx, int dy) {
    this.x += dx; this.y += dy;
  }
  @Override
  public String toString() {
    return(this.x + ", " + this.y);
  }
}

class ThickPixel extends Pixel {
  int thickness;
  ThickPixel(int x, int y, int t) {
    super(x, y);
    thickness = t;
  }
  @Override
  public String toString() {
    return super.toString() + ", " + thickness;
  }
  void thicken(int i) {
    thickness += i;
  }
  public static void main(String[] args) {
    ThickPixel a = new ThickPixel(3, 5, 1);
    System.out.println(a); // 3, 5, 1
    a.translate(5,-2); System.out.println(a); // 8, 3, 1
    a.thicken(5); System.out.println(a); // 8, 3, 6
  }
}
```

Dans l'exécution d'un constructeur, le constructeur de la classe de base est exécuté en premier. Par défaut, c'est le constructeur sans argument de la classe de base. On remonte récursivement jusqu'à la classe `Object`.
Dans un constructeur d'une classe dérivée, l'appel d'un autre constructeur de la classe de base se fait au moyen de `super(...)`. Cette instruction doit être la première dans l'écriture du constructeur de la classe dérivée. En d'autres termes, si cette instruction est absente, c'est l'instruction `super()` qui est exécutée en premier.

This et super

- `this` et `super` sont des références sur l'objet courant.
- `super` désigne l'objet courant avec le type père. Il indique que la méthode invoquée ou le membre d'objet désigné doit être recherché dans la classe de base. Il y a des restrictions d'usage (`f(super)` est interdit).
- L'usage de `this` et `super` est spécial dans les constructeurs.



## Destruction des objets

La destruction des objets est effectuée :

- automatiquement par le ramasse-miettes;
- de façon asynchrone, avec un processus léger de basse priorité;
- la méthode `finalize()` permet de spécifier des actions à effecruer au moment de la destruction de l'objet (opérations de nettoyage, fermeture de fichiers ...).

L'appel au ramasse-miettes peut être forcé par l'appel `System.gc()`.

```
Class Disque {
  protected void finalize(){
    System.out.println("Disque detruit"); }
}
```



## Points et disques : exemple de délégation

Il ne faut pas confondre dérivation et délégation.

```
class Pixel {
  int x, y;
  Pixel (int x, int y) {
    this.x = x; this.y = y;
  }
  void translate(int dx, int dy) {
    this.x += dx; this.y += dy;
  }
  @Override public String toString() {
    return(this.x + ", " + this.y);
  }
}
class Disk {
  Pixel center;
  int radius;
  Disk(int x, int y, int radius) {
    center = new Pixel(x,y);
    this.radius = radius;
  }
  @Override
  public String toString() {
    return center.toString() + " ," + radius;
  }
  void translate(int dx, int dy) {
    center.translate(dx, dy); // délégation
  }
}
class DiskTest {
  public static void main(String[] args) {
    Disk d = new Disk(3, 5, 1);
    System.out.println(d); // 3, 5 ,1
    d.translate(5,-2);
    System.out.println(d); // 8, 3 ,1
  }
}
```



## Redéfinition

Une méthode *redéfinie* est une méthode d'une classe dérivée qui a même *signature* que la méthode mère, c'est-à-dire même :

- nom;
- suite des types des paramètres;
- type de retour (ou un sous-type de celui-ci (jdk 1.5)).

De plus :

- les exceptions levées doivent aussi être levées par la méthode de la classe mère, et être au moins aussi précises.
- la visibilité de la méthode doit être au moins aussi bonne que celle de la méthode de la classe mère (pas de restriction de visibilité).

En cas de redéfinition, la méthode invoquée est déterminée par le mécanisme de *liaison tardive* (voir juste après).
En cas de redéfinition, la méthode de la classe de base n'est plus accessible à partir de l'objet appelant : la méthode de la classe de base est *masquée*.

*Exemple*

```
public class Alpha{
  void essai(Alpha a){
    System.out.println("alpha");
  }
}

public class Beta extends Alpha {
    void essai(Beta b){
      System.out.println("beta");
    }
    public static void main(String[] args) {
        Beta b = new Beta();
        Alpha c = new Beta();
        b.essai(c);
    }
}

public class Gamma extends Beta {
     @Override
     void essai(Alpha a){
       System.out.println("gamma");
     }
    public static void main(String[] args){
        Beta d = new Gamma();
        Alpha e = new Gamma();
        d.essai(e);
    }
}
```

On obtient

```
$ java Beta
alpha
$ java Gamma
gamma
```



## Mécanisme de détermination de la méthode appelée I

Pré-sélection au niveau statique

- Au niveau statique, le compilateur ne connaît pas la nature de l'objet, mais le type de sa référence.

```
public class Alpha{
  void essai(Alpha a){   (1)
    System.out.println("alpha");
}}
public class Beta extends Alpha {
    void essai(Beta b){  (2)
      System.out.println("beta");
    }
    public static void main(String[] args) {
        Beta b = new ??();
        Alpha c = new ??();
        Beta ◁ essai( Alpha );
}}
```

- Il pré-sélectionne une méthode visible, de profil compatible avec l'appel. Les types des arguments doivent être compatibles en tenant compte de l'autoboxing et du sous-typage. Ici : `essai(Alpha)`.

Au niveau dynamique

- Chaque objet a un pointeur sur une table de pointeurs de fonctions. Pour un objet Beta,
  - `essai(Alpha)`→ (1),
  - `essai(Beta)`→ (2),
- La méthode appelée est alors déterminée par la nature de l'objet appelant (***liaison tardive***) en fonction des méthodes pré-sélectionnées et de cette table : ici (1).



## Mécanisme de détermination de la méthode appelée II

Pré-sélection au niveau statique

- Au niveau statique, le compilateur voit

```
public class Alpha{
  void essai(Alpha a){   (1)
    System.out.println("alpha");
}}
public class Beta extends Alpha {
    void essai(Beta b){  (2)
      System.out.println("beta");
    }
}}
public class Gamma extends Beta {
      @Override void essai(Alpha a){ (3)
       System.out.println("gamma");
      }
    public static void main(String[] args){
        Beta d = new ??();
        Alpha e = new ??();
        Beta ◁ essai(Alpha);
}}
```

- Ici `essai(Alpha)` et `essai(Beta)` sont visibles (On voit statiquement au-dessus de Beta). Seule `essai(Alpha)` est compatible.

Au niveau dynamique

- L'objet `d` est un Gamma. Sa table est
  - `essai(Alpha)`→ (3),
  - `essai(Beta)`→ (2),
- La méthode appelée est alors déterminée par la nature de l'objet appelant, ici (3).



## Mécanisme de détermination de la méthode appelée III

```
public class Alpha{
  void essai(Alpha a){   (1)
    System.out.println("alpha");
}}
public class Beta extends Alpha {
    void essai(Beta b){  (2)
      System.out.println("beta");
    }
}}
public class Gamma extends Beta {
      @Override void essai(Alpha a){ (3)
       System.out.println("gamma");
      }
    public static void main(String[] args){
        Beta d;
        if (Math.random()> 0.5) d = new Gamma();
        else d = new Beta();
        Alpha e = new Gamma();
        d.essai(e);
}}
```

- Au niveau statique, seule `essai(Alpha)` est compatible.
- Si l'objet `d` est un Gamma. Sa table est
  - `essai(Alpha)`→ (3),
  - `essai(Beta)`→ (2).
- Si l'objet `d` est un Beta. Sa table est
  - `essai(Alpha)`→ (1),
  - `essai(Beta)`→ (2).
- La méthode appelée est (1) ou (3). On ne peut le savoir (statiquement).

## Mécanisme de détermination de la méthode appelée IV

```
class A{
 void f(A o) { System.out.print("1 ");} (1)
}
class B extends A{
 void f(A o) { System.out.print("2 ");} (2)
 void f(B o) { System.out.print("3 ");} (3)
}
class Test{
 public static void main(String[] args){
  A a = new A();
  A ab = new B();
  B b = new B();
  ab.f(b);
}
```

Pré-sélection au niveau statique

- Au niveau statique, le compilateur ne voit que `f(A)`, qui est compatible avec le sous-typage des arguments.

Au niveau dynamique

- L'objet `ab` est un `B`. Sa table est
  - `f(A)`→ (2),
  - `f(B)`→ (3),
- La méthode appelée est (2).



## Le transtypage

Le transtypage

- modifie le type de la référence à un objet;
- n'affecte que le traitement des références : *ne change jamais le type de l'objet*;
- est implicite ou explicite.

On a vu que le sous-typage est automatique et qu'il coïncide avec la dérivation en Java : une variable de type $B$ peut contenir implicitement une référence à un objet de toute classe dérivée $D$ de $B$.
Commentaires :

- Cette règle s'applique aussi si $B$ est une interface et $D$ implémente $B$;
- La relation est transitive;
- Cette règle s'applique aussi aux paramètres d'une méthode : pour

  ```
  C f(B b) {...}
  ```

  on peut faire l'appel `f(d)`, avec `d` de classe $D$
- Cette règle s'applique aussi aux valeurs de retour d'une méthode : pour la méthode ci-dessus, on peut écrire

  ```
  r = f(b);
  ```

  si `r` est d'une superclasse de `C`.



Le transtypage explicite d'une référence n'est valide que si l'objet sousjacent est d'une classe dérivée.

```
// sous-typage automatique
Pixel p = new ThickPixel(5, 7, 1);
ThickPixel q;
q = p;           // erreur
// transtypage explicite ou cast
q = (ThickPixel) p;  // ok
```

**En résumé**, types et classes ne sont pas la même chose.

- "Variables have type, objects have class";
  - un objet ne change jamais de classe;
  - les références peuvent changer de type;
- la vérification des types est statique (à la compilation);
- la détermination de la méthodes à invoquer est surtout dynamique (à l'exécution). Elle comporte une part statique.

**Intérêt du transtypage**

- permet de forcer l'appel d'une méthode en changeant le type d'un argument.
- permet le *polymorphisme*. Des objets de natures différentes dérivant d'une même classe `A` peuvent être typés par cette classe. Par le mécanisme de liaison tardive, la méthode appelée en cas de redéfinition dans les sous-classes, est la méthode de l'objet. Le fait qu'ils soient typés par `A` ne gêne pas.
- Ceci permet l'*encapsulation* : on donne le nom de la méthode dans `A` (en général une interface), l'implémentation se fait dans les sous-classes et peut être cachée.



- Ceci permet l'*abstraction*. Une méthode générique peut être utilisée dans `A`. Cette méthode est spécialisée dans les classes dérivées.

## Héritage : interfaces

Une *interface* est une classe

- n'a que des méthodes abstraites et tacitement publiques;
- et n'a que des données `static` immuables (`final`).

Une interface sert à spécifier des méthodes qu'une classe doit avoir, sans indiquer comment les réaliser.

Une *classe abstraite* est une classe

- peut avoir des méthodes concrètes ou abstraites.

Une méthode abstraite est déclarée mais non définie.

Une classe abstraite sert en général à commencer les implémentations (parties communes aux classes qui en dériveront).

On ne peut créer d'instance que d'une classe concrète. Toutes les méthodes doivent être définies dans la classe ou les classes mères.

Pas de `new AbstractShape()`.

L'interface est le point ultime de l'abstraction. C'est un style de programmation à encourager.



## Program to Interfaces

### Rectangles et ellipses

Une classe `Rectangle` serait :

```
class Rectangle {
  double width, height;
  Rectangle(double width, double height) {
    this.width = width; this.height = height;
  }
  double getArea() {return width*height;}
  String toStringArea() {
    return "aire = "+getArea();
  }
}
```

et une classe `Ellipse` serait

```
class Ellipse {
  double width, height;
  Ellipse(double width, double height) {
    this.width = width; this.height = height;
  }
  double getArea(){
    return width*height*Math.PI/4;}
  String toStringArea() {
    return "aire = "+getArea();
  }
}
```

Ces classes ont une *abstraction commune*, qui

- *définit* les méthodes de même implémentation;
- *déclare* les méthodes communes et d'implémentation différentes.



L'*interface* `Shape`

```
interface Shape {
  double getArea();
  String toStringArea();
}
```

La classe *abstraite* `AbstractShape` définit l'implémentation de `toStringArea`

```
abstract class AbstractShape implements Shape {
  double width, height;
  AbstractShape(double width, double height) {
    this.width = width; this.height = height;
  }
  public String toStringArea() {
    return "aire = " + getArea();
  }
}
```

Les méthodes *abstraites* sont implémentées dans chaque classe concrète.

```
class Rectangle extends AbstractShape{
  Rectangle(double width, double height) {
    super(width, height);
  }
  public double getArea() {return width*height;}
}
```

et

```
class Ellipse extends AbstractShape {
  Ellipse(double width, double height) {
    super(width, height);
  }
  public double getArea() {return Math.PI*width*height/4;}
}
```



On s'en sert par exemple dans :

```
Shape r = new Rectangle(6,10);
Shape e = new Ellipse(3,5);
System.out.println(r.toStringArea());
System.out.println(e.toStringArea());
```

ou dans :

```
Shape[] tab = new Shape[5];
tab[0] = new Rectangle(6,10);
tab[1] = new Ellipse(3,5);
...
for (Shape s:tab)
   System.out.println(s.toStringArea());
```

## Instanceof

On peut tester le type d'un objet à l'aide de `instanceof` :

```
Shape s = new Rectangle(6,10);

if (s instanceof Object) {...} // vrai
if (s instanceof Shape) {...} // vrai
if (s instanceof Rectangle) {...} // vrai
if (s instanceof Ellipse) {...} // faux
```

L'usage de `instanceof` est restreint à des cas bien particuliers. Il ne doit pas se substituer au polymorphisme.



## Exemples d'interface : les matrices suite

```
interface Matrix {
  Matrix add(Matrix a);
  void setAt(int i, int j, int value);
  void transpose();
}
```

Deux implémentations, à savoir des matrices générales et des matrices symétriques, se partagent une classe abstraite commune.

```
//Abstract class for matrices of ints.
public abstract class AbstractMatrix implements Matrix{
   int[][] m;
   public AbstractMatrix(int dim) {
       this(dim,0);
   }
   public AbstractMatrix(int dim, int n) {
      m = new int [dim][dim];
      for (int i = 0; i < dim; i++)
       for (int j = 0; j < dim; j++)
          m[i][j] = n;
   }
   @Override public String toString(){
     StringBuilder sb = new StringBuilder();
     for (int[] t : m){
          for (int x : t){
             sb.append(x);
             sb.append(" ");
          }
        sb.append("\n");
     }
     return sb.toString();
   }
}
```



Les classes spécifiques se contentent d'implémenter les autres méthodes:

```
public class GeneralMatrix extends AbstractMatrix{
  public GeneralMatrix(int dim){
    super(dim);
  }
  public GeneralMatrix(int dim, int n){
    super(dim,n);
  }
  public void transpose() {
    int dim = m.length;
    for (int i = 0; i < dim; i++)
      for (int j = i+1; j < dim; j++){
        int t = m[i][j];
        m[i][j] = m[j][i];
        m[j][i]=t;
    }
  }
  public void setAt(int i, int j, int value) {
     m[i][j] = value;
  }
  public Matrix add(Matrix a){
    int n = m.length;
    Matrice s = new GeneralMatrice(n);
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        s.setAt(i, j, m[i][j] + ((AbstractMatrice)a).m[i][j]);
    return s;
  }
}

public class SymmetricMatrix extends AbstractMatrix{
   public SymmetricMatrix(int dim) {
       super(dim);
   }
   public SymmetricMatrix(int dim, int n){
       super(dim,n);
   }
```



```
   public void transpose() {}
   public void setAt(int i, int j, int value) {
      m[i][j] = m[j][i] = value;
   }
   public Matrix add(Matrix a){
    if (a instanceof SymmetricMatrix){
      int n = m.length;
      SymmetricMatrix sa = (SymmetricMatrix) a;
      Matrix s = new SymmetricMatrix(n);
      for (int i = 0; i < n; i++)
       for (int j = 0; j < n; j++)
        s.setAt(i, j, m[i][j] + sa.m[i][j]);
      return s;
    }
     else return a.add(this);
   }

  public static void main(String[] args) {
    Matrix a = new GeneralMatrix(3,12);
    a.setAt(1,2,0);
    System.out.print(a);
    Matrix b = new SymmetricMatrix(3,14);
    b.setAt(1,2,0);
    System.out.print(b);
    Matrix s = a.add(b); System.out.print(s);
    Matrix t = b.add(b); System.out.print(t);
  }
}
```

Résultat :

```
12 12 12    14 14 14    26 26 26    28 28 28
12 12 0     14 14 0     26 26 0     28 28 0
12 12 12    14 0 14     26 12 26    28 0 28
```

*Autre exemple : application d'une méthode à tous les éléments d'un conteneur.*

- On encapsule une méthode dans une interface :

```
interface Function {
  int applyIt(int n);
}
```

- On encapsule la fonction `mapcar` dans une autre interface :

```
interface Map {
  void map(Function f);
}
```

- Une classe qui peut réaliser un `map` implémente cette interface :

```
class Tableau implements Map {
  int[] a;
  Tableau(int n) {
    a = new int[n];
    for (int i = 0; i < n; i++)
      a[i] = i + 1;
  }
  public void map(Function f) {
    for (int i = 0; i < a.length; i++)
      a[i] = f.applyIt(a[i]);
  }
  public String toString() {...}
}
```



- Un exemple de fonction :

```
class Carre implements Function {
  public int applyIt(int n) {
    return n*n;
  }
}
```

- Un exemple d'utilisation:

```
public class TestMap {
  public static void main(String args[]) {
    Tableau t = new Tableau(10);
    Function square = new Carre();
    System.out.print(t);
    t.map(square);
    System.out.print(t);
  }
}

1 2 3 4 5 6 7 8 9 10
1 4 9 16 25 36 49 64 81 100
```



## Exceptions

Voici une classe pile contenant des int, implémentée par un tableau.

```
public class Stack{
  static final int MAX=4;
  int height = 0;
  int[] table = new int[MAX];

  public boolean isEmpty() {
    return height == 0;
  }

  public boolean isFull() {
    return height == MAX;
  }

  public void push(int item) {
    table[height++] = item;
  }

  public int peek() {
    return table[height-1];
  }

  public int pop() {
    --height;
    return table[height];
  }
}
```



Il y a *débordement* lorsque l'on fait

- `peek` pour une pile vide;
- `pop` pour une pile vide;
- `push` pour une pile pleine.

Une pile ne peut pas proposer de solution en cas de débordement, mais elle doit *signaler* (et interdire) le débordement. Cela peut se faire par l'usage d'une *exception*.

Une *exception* est un objet d'une classe qui étend la classe `Exception`.

```
java.lang.Object
 |_ java.lang.Throwable
      |_java.lang.Error
      |_java.lang.Exception
          |_java.lang.ClassNotFoundException
          |_ ...
          |_java.lang.RuntimeException
              |_java.lang.NullPointerException
              |_java.lang.UnsupportedOperationException
              |_ ...
```

Pour les piles, on peut définir par exemple une nouvelle exception.

```
class StackException extends Exception {}
```

En cas de débordement, on *lève* une exception, par le mot `throw`. On doit signaler la possible levée dans la déclaration par le mot `throws`.

Par exemple :

```
void push(int item) throws StackException {
  if (isFull())
    throw new StackException("Pile pleine");
  table[height++] = item;
}
```

L'effet de la levée est

- la propagation d'un objet d'une classe d'exceptions qui est en général créé par `new`;
- la sortie immédiate de la méthode;
- la remontée dans l'arbre d'appel à la recherche d'une méthode qui *capture* l'exception.

La *capture* se fait par un bloc `try / catch`. Par exemple,

```
...
Stack s = new Stack();
try {
  System.out.println("top = "+p.peek());
} catch(StackException e) {
  System.out.println(e.getMessage());
}
...
```

- Le bloc `try` lance une exécution contrôlée.
- En cas de levée d'exception dans le bloc `try`, ce bloc est quitté immédiatement, et l'exécution se poursuit par le bloc `catch`.
- Le bloc `catch` reçoit en argument l'objet créé lors de la levée d'exception.
- Plusieurs `catch` sont possibles, et le premier dont l'argument est du bon type est exécuté. Les instructions du bloc `finally` sont exécutées dans tous les cas.



```
try { ... }
catch (Type1Exception e) { .... }
catch (Type2Exception e) { .... }
catch (Exception e) { .... }
     // cas par defaut, capture les
     // exceptions non traitees plus haut
finally {....} // toujours execute
```

Exemple

```
try { ... }
catch (Exception e) { .... }
catch (StackException e) { .... }
// le deuxième jamais exécuté
```

Une levée d'exception se produit lors d'un appel à `throw` ou d'une méthode ayant levé une exception. Ainsi l'appel à une méthode pouvant lever une exception doit être :

- ou bien être contenu dans un bloc `try / catch` pour capturer l'exception;
- ou bien être dans une méthode propageant cette classe d'exception (avec `throws`).

Les exceptions dérivant de la classe `RuntimeException` n'ont pas à être capturées.



Voici une interface de pile d'int, et deux implémentations.

```
interface Stack {
  boolean isEmpty ();
  boolean isFull();
  void push(int item) throws StackException;
  int peek() throws StackException;
  int pop() throws StackException;
}

class StackException extends Exception {
  StackException(String m) {super(m);}
}
```

Implémentation par tableau

```
public class ArrayStack implements Stack {
  static final int MAX=4;
  private int height = 0;
  private int[] table = new int[MAX];

  public boolean isEmpty() {
    return height == 0;
  }
  public boolean isFull() {
    return height == MAX;
  }
  public void push(int item) throws StackException {
    if (isFull())
      throw new StackException("Pile pleine");
    table[height++] = item;
  }
  public int peek() throws StackException{
    if (isEmpty())
      throw new StackException("Pile vide");
    return table[height-1];
  }
  public int pop() throws StackException{
    if (isEmpty())
```



```
      throw new StackException("Pile vide");
    --height;
    return table[height];
  }
}
```

Implémentation par listes chaînées (classe interne `Liste`).

```
public class LinkedStack implements Stack{
  private Liste head = null;
  class Liste {
    int item;
    Liste next;
    Liste(int item, Liste next) {
      this.item = item; this.next = next;
    }
  }
  public boolean isEmpty() {
    return head == null;
  }
  public boolean isFull() {
    return false;
  }
  public void push(int item) throws StackException{
    head = new Liste(item, head);
  }
  public int peek() throws StackException{
    if (isEmpty())
      throw new StackException("Pile vide");
    return head.item;
  }
  public int pop() throws StackException{
    if (isEmpty())
      throw new StackException("Pile vide");
    int i = head.item;
    head = head.next;
    return i;
 }}
```

Usage:

```
public class Test{
  public static void main(String[] args) {
    Stack s = new ArrayStack(); //par table
    try {
      s.push(2); s.peek();
      s.pop(); // ca coince
      s.pop(); // jamais atteint
    }
    catch(StackException e) {
      System.out.println(e.getMessage());
      e.printStackTrace();
    }
    Stack t = new LinkedStack();  //par liste
    try {
      t.push(2); t.push(5);
      t.pop(); t.pop();
      System.out.println(t.peek()); // ca coince
    }
    catch(StackException e) {
      System.out.println(e.getMessage());
      e.printStackTrace();
    }
  }
}
```

On obtient

```
Pile vide
StackException: Pile vide
        at ArrayStack.pop(ArrayStack.java:24)
        at Test.main(Test.java:6)
Pile vide
StackException: Pile vide
        at LinkedStack.peek(LinkedStack.java:21)
        at Test.main(Test.java:17)
```



## Piles génériques

```
interface Stack<E>{
  boolean isEmpty ();
  boolean isFull();
  void push(E item) throws StackException;
  E peek() throws StackException;
  E pop() throws StackException;
}
public class LinkedStack<E> implements Stack<E>{
  private Liste<E> head = null;
  class Liste<E>{
    E item;
    Liste<E> next;
    Liste(E item, Liste<E> next){
      this.item = item; this.next = next;
  }}
  public boolean isEmpty() {
    return head == null;
  }
  public boolean isFull() {
    return false;
  }
  public void push(E item) throws StackException{
    head = new Liste<E>(item, head);
  }
  public E peek() throws StackException{
    if (isEmpty())
      throw new StackException("Pile vide");
    return head.item;
  }
  public E pop() throws StackException{
    if (isEmpty())
      throw new StackException("Pile vide");
    E i = head.item;
    head = head.next;
    return i;
  }}
```



## Utilisation

```
public class Test{
  public static void main(String[] args) {
    Stack<Integer> s = new LinkedStack<Integer>();
    try {
      s.push(2); s.pop();
      s.push(3L); //ne compile pas
      s.pop(); // ca coince
      s.pop(); // jamais atteint
    }
    catch(StackException e) {
      System.out.println(e.getMessage());
      e.printStackTrace();
    }
    Stack<Boolean> t = new LinkedStack<Boolean>();
    try {
      t.push(true); t.push(false); //test à la compilation
      t.pop(); t.pop();
      System.out.println(t.peek()); // ca coince
    } catch(StackException e) {
      System.out.println(e.getMessage());
      e.printStackTrace();
    }
  }
}
```



## Abrasure

- Les types paramétrés sont utilisés pour permettre de vérifier à la compilation des concordances de types tout en gardant la généricité.
- Le compilateur transforme ensuite les types paramétrés par le mécanisme d'abrasure.
- Pour la machine virtuelle, les types paramétrés n'existent pas.

```
public class LinkedStack<E> implements Stack<E>{
  private Liste<E> head = null;
  class Liste<E>{
    E item;
    Liste next;
    Liste(E item, Liste<E> next){
      this.item = item; this.next = next;
  }}
  public void push(E item) throws StackException{
    head = new Liste<E>(item, head);
  }
```

est changé en

```
public class LinkedStack implements Stack{
  private Liste head = null;
  class Liste{
    Object item;
    Liste next;
    Liste(Object item, Liste next){
      this.item = item; this.next = next;
  }}
  public void push(Object item) throws StackException{
    head = new Liste(item, head);
  }
```

et

```
public class Test{
  public static void main(String[] args) {
    Stack<Integer> s = new LinkedStack<Integer>();
    try {
      s.push(2); s.pop();
      s.push(3L); //ne compile pas
      s.pop(); // ca coince
      s.pop(); // jamais atteint
    }
```

est changé en

```
public class Test{
  public static void main(String[] args) {
    Stack s = new LinkedStack();
    try {
      s.push((Integer)2); s.pop();
      s.push((Integer)3L);
       // erreur de compilation ici
      (Integer)s.pop();
      (Integer)s.pop();
    }
```



## Un pattern de création : les fabriques

Une ***fabrique*** est une classe dont des méthodes ont en charge la construction d'objets d'une autre classe.

Réalisation : une méthode

```
Box createBox() {return new Box();}
```

Exemple : On cherche une méthode `testVersion()` qui permet de remplacer le corps de la méthode main de l'exemple par deux appels. Voici quelques variantes.

```
public static void testVersion1(Stack s) {
  try {
    s.push(2); s.push(5);
    s.pop(); s.pop();
    System.out.println(s.peek());
  }
  catch(StackException e) {
    System.out.println(e.getMessage());
  }
}
```

utilisé avec

```
testVersion1(new ArrayStack());
testVersion1(new LinkedStack());
```

Si l'on veut créer à l'intérieur de la méthode de test:

```
public static void testVersion2(boolean version) {
  Stack s;
  if (version)
    s = new ArrayStack();
  else
    s = new LinkedStack();
  ...
}
```



utilisé avec

```
testVersion2(true);
testVersion2(false);
```

On peut "délocaliser" la création en une méthode de fabrique.

```
public static Stack createStack(boolean version) {
  if (version) return new ArrayStack();
  else return new LinkedStack();
}

public static void testVersion3(boolean version) {
  Stack s;
  s = createStack(version);
  ...
}
```

utilisé avec

```
testVersion3(true);
testVersion3(false);
```

On peut enfin transmettre un ***descripteur de classe***:

```
public static <T> T factory(Class<T> clazz)
throws IllegalAccessException, InstantiationException{
    return clazz.newInstance();
}
```

La méthode T `newInstance()` de la classe `Class`<T> est une méthode de fabrique. On utilise cette méthode avec

```
Stack s;
try {
  s = factory(ArrayStack.class);
  s = factory(LinkedStack.class);
}
catch(IllegalAccessException e) {}
catch(InstantiationException e) {}
```



## Paquetages

Paquetage (*package*): un mécanisme de groupement de classes.

- Les classes d'un paquetage sont dans un même répertoire décrit par le nom du paquetage.
- Le nom est relatif aux répertoires de la variable d'environnement `CLASSPATH`.
- Les noms de paquetage sont en minuscule.

Par exemple, le paquetage `java.awt.event` est dans le répertoire `java/awt/event` (mais les classes Java sont zippées dans les archives).

Importer `java.awt.event.*` signifie que l'on peut *nommer* les classes dans ce répertoire par leur nom local, à la place du nom absolu. Cela ne concerne que les fichiers `.class` et non les répertoires contenus dans ce répertoire.

Exemple :

`class MyApplet extends Applet` *non trouvée*

`class MyApplet extends java.applet.Applet` *ok*

`import java.applet.Applet;`
`class MyApplet extends Applet` *ok*

`import java.applet.*;`
`class MyApplet extends Applet` *ok*

Pour faire son propre paquetage : on ajoute la ligne

`package repertoire;`

en début de chaque fichier `.java` qui en fait partie. Le fichier `.java` doit se trouver dans un répertoire ayant pour nom `repertoire`.

Par défaut, le paquetage est sans nom (`unnamed`), et correspond au répertoire courant.
Si une même classe apparaît dans deux paquetages importés globalement, la classe utilisée doit être importée explicitement.



## Visibilité et paquetages

*Visibilité des classes et interfaces*

Une classe ou une interface qui est déclarée `public` est accessible en dehors du paquetage.

Si elle n'est pas déclarée `public`, elle est accessible à l'intérieur du même paquetage, mais cachée en dehors.

Il faut déclarer publiques les classes utilisées par les clients utilisant le paquetage et cacher les classes donnant les détails d'implémentation.
Ainsi, quand on change l'implémentation, les clients ne sont pas concernés par les changements puisqu'ils n'y ont pas accès.



## Différences entre Java et C++

Java n'a pas

- de préprocesseur (`#define` ou `#include`),
- de fichier en-tête séparés (`.h` et `.c`),
- de variable ou fonctions globales,
- de valeurs par défaut dans les fonctions,
- de pointeurs (pas de pointeurs de fonction),
- de surcharge d'opérateurs,
- de passage d'argument par recopie,
- d'allocation statique de mémoire.

Java a

- une méthode `finalize()` appelée à la destruction d'objets,
- une classe universelle (`Object`) et des types paramétrés,
- un héritage simple, mais la possibilité d'implémenter un nombre quelconque d'interfaces,
- la possibilité de déterminer le type d'un objet à l'exécution (`instanceof`),
- des possibilités d'introspection : `java.lang.Class`,
- une grande robustesse par vérification : `IndexOutOfBounds-Exception`, `ClassCastException`, etc.
- de nombreuses classes utilitaires prédéfinies.



## Programmation des listes chaînées

Une liste est une suite d'objets.

- Comme séquence $(a_1, \ldots, a_n)$, elle se programme itérativement.
- Comme structure imbriquée
$$(a_1, (a_2, (\cdots (a_n, ()) \cdots)))$$
elle se définit récursivement.

Une ***cellule*** est

- soit une ***cellule vide***
- soit un ***cons*** d'un objet et d'une cellule

Une ***liste*** contient une cellule initiale. Ceci conduit à une classe pour les listes, avec trois interfaces, pour les cellules, les cellules cons et les cellules vides.

```
public class Liste {
   Cell init ;
   public Liste () {init = new ConcreteNil();}
   public int length() {return init.length();}
}

interface Cell {
  int length();
}

interface Cons extends Cell {
  Object getElem();
  void setElem(Object o);
  Cell getNext();
```

```
  void setNext(Cell next);
}

interface Nil extends Cell {}
```

L'implémentation se fait naturellement:

```
class ConcreteCons implements Cons{
  private Object o;
  private Cell next;
  ConcreteCons(Object o, Cell next){
    this.o = o;
    this.next = next;
  }
  public Object getElem(){
    return o;
  }
  public void setElem(Object o){
    this.o = o;
  }
  public Cell getNext(){
    return next;
  }
  public void setNext(Cell next){
    this.next = next;
  }
  public int length(){
    return next.length()+1;
  }
}

class ConcreteNil implements Nil {
  public int length(){
    return 0;
  }
}
```



Usage:

```
public class ListeTest{
  public static void main(String[] args) {
    Liste l = new Liste();
    l.init = new ConcreteCons(1,
             new ConcreteCons(2,new ConcreteNil()));
    System.out.println("liste de longueur "+ l.length());
  }
}
```

Plusieurs appels à `ConcreteNil()` créent des cellules vides différentes. Pour l'éviter, on change :

```
public class ConcreteNil implements Nil{
  private static Cell nulle = new ConcreteNil();
  private ConcreteNil() {}
  public static Cell getNil() {return nulle;}
  public int length(){return 0;}
}
```

avec bien sûr

```
    l.init = new ConcreteCons(1,
             new ConcreteCons(2, ConcreteNil.getNil()));
```



## Un pattern de création : singleton

Une classe ***singleton*** est une classe qui ne peut avoir qu'une seule instance.

Réalisation

- un attribut privé statique `instance` désignant l'instance;
- une méthode publique de création qui teste si l'instance existe déjà;
- un constructeur privé.

Exemple

```
public class ConcreteNil implements Nil {
  private static ConcreteNil instance = null;
  private ConcreteNil() {}
  public static Cell getNil(){
    if (instance == null)
      instance = new ConcreteNil();
    return instance;
  }
  public int length(){return 0;} // autres méthodes
}
```



## Listes chaînées génériques

```
public class Liste<E> {
   Cell<E> init ;
   public Liste() { init = new ConcreteNil<E>();}
   public int length() { return init.length();}
   public Iterator<E> iterator() { ... }
}

interface Cell<E> {
  int length();
}

interface Cons<E> extends Cell<E> {
  E getElem();
  void setElem(E o);
  Cell<E> getNext();
  void setNext(Cell<E> next);
}

interface Nil<E> extends Cell<E> {}
```

L'implémentation se fait naturellement:

```
class ConcreteCons<E> implements Cons<E> {
  private E o;
  private Cell<E> next;
  ConcreteCons(E o, Cell<E> next){
    this.o = o;
    this.next = next;
  }
  public E getElem(){
    return o;
  }
  public void setElem(E o){
    this.o = o;
```

```
    }
    public Cell<E> getNext(){
      return next;
    }
    public void setNext(Cell<E> next){
      this.next = next;
    }
    public int length(){
      return next.length()+1;
    }
  }

  class ConcreteNil<E> implements Nil<E> {
    public int length(){
      return 0;
    }
  }
```

Usage:

```
  public class ListeTest {
    public static void main(String[] args) {
      Liste<Integer> l = new Liste<Integer>();
      l.init = new ConcreteCons<Integer>
               (1, new ConcreteCons<Integer>
               (2,new ConcreteNil<Integer>()));
      System.out.println("liste de longueur "+ l.length());
    }
  }
```



# 3

## Les classes fondamentales

1. Présentation des API
2. La classe `java.lang.Object`
   mère de toutes les classes
3. Les chaînes de caractères
4. Outils mathématiques
5. Ensembles structurés, itérateurs et comparateurs
6. Introspection



## Les API

Les API (Application Programming Interface) forment l'interface de programmation, c'est-à-dire l'ensemble des classes livrées avec Java.

| | |
|---|---|
| `java.lang` | classes de base du langage |
| `java.io` | entrées / sorties |
| `java.util` | ensemble d'outils : les classes très "util" |
| `java.net` | classes réseaux |
| `java.applet` | classes pour les appliquettes |
| `java.awt` | interfaces graphiques (Abstract Windowing Toolkit) |
| `javax.swing` | interfaces graphiques |
| ... | |

et de nombreuses autres.



## La classe `java.lang.Object`

```
protected Object       clone()
                       throws CloneNotSupportedException
public boolean         equals(Object obj)
protected void         finalize()
public final Class<?>  getClass()
public int             hashCode()
public String          toString()
public final void      notify()
public final void      notifyAll()
public final void      wait()
public final void      wait(long timeout)
public final void      wait(long timeout, int nanos)
```

### Affichage d'un objet et hashcode

La méthode `toString()` retourne la représentation d'un objet sous forme de chaîne de caractères (par défaut le nom de la classe suivi de son *hashcode*) :

```
System.out.println(new Integer(3).toString());
//affiche 3
System.out.println(new Object().toString());
//affiche java.lang.Object@1f6a7b9
```

La valeur du hashCode peut être obtenue par la méthode `hashCode()` de la classe `Object`.



## Le clonage

Le clonage est la construction d'une copie d'un objet.

La classe `Object` contient une méthode
`Protected Object clone() throws CloneNotSupportedException`.

- Quand elle est appelée sur un objet d'une classe qui implémente l'interface `Cloneable`, elle crée une copie de l'objet du même type. On a une copie superficielle de l'objet : les attributs de l'objet sont alors recopiés.
- Quand elle est appelée sur un objet d'une classe qui n'implémente pas l'interface `Cloneable`, elle lève l'exception `CloneNotSupportedException`.
- La class `Object` n'implémente pas l'interface `Cloneable` !

Exemple:

```
public class Pixel implements Cloneable {
  private int x, y;

  public Pixel (int x, int y) {
    this.x = x; this.y = y;
  }
  public int getX(){
    return this.x
  }
  public void setX(int x){
    this.x=x;
  }

  @Override public String toString() {
    return(this.x + ", " + this.y);
  }
```



```
  public static void main(String[] args)
    throws CloneNotSupportedException
  {
    Pixel a = new Pixel(3,5);
    Pixel b = (Pixel) a.clone(); // méthode clone() de Object
    b.setX(a.getX() + 1);
    System.out.println(a); // 3 5
    System.out.println(b); // 4 5
  }
}
```

Si on veut une classe clonable et une classe dérivée non clonable, la classe dérivée implémente `Cloneable` mais on lève une exception dans l'écriture de `clone()`.

### Exemple de clonage

Le clonage par défaut est superficiel.

```
public class Stack implements Cloneable {
   int height;
   Integer[] table;

   public Stack (int max) {
     height = 0;
     table = new Integer[max];
   }

  public void push(Integer item) {
    table[height++]=item;
  }

  public Integer pop() {
    return table[--height];
  }
```



```
  public Object clone() throws CloneNotSupportedException {
    Stack instance = (Stack) super.clone();
    instance.table = (Integer[]) table.clone();
    return instance;
  }
}

public class Test{
  public static void main(String[] args) {
    Stack s = new Stack(2);
    s.push(5);
    s.push(6);
    try {
      Stack t = (Stack) s.clone();
      System.out.println(t.pop());  // 6
      System.out.println(t.pop());  // 5
    } catch (CloneNotSupportedException e) {}
  }}
```

Remarquer l'utilisation de `super.clone()` qui appelle `clone()` de `Object` crée toujours un objet du bon type. L'appel à `clone()` sur un objet d'une classe dérivée de `Stack` serait incorrect si on avait utilisé `new Stack()`.
Exemple

```
public class PriceStack extends Stack{
  int price;

  public PriceStack (int max) {
    super(max);
    price = 0;
  }

  public Object clone() throws CloneNotSupportedException   {
    PriceStack instance = (PriceStack) super.clone();
    // instance.price = price;  (facultatif)
    return instance;
  }}
```

## Égalité entre objets et hashCode

La méthode `equals()` de la classe `Object` détermine si deux objets sont égaux. Par défaut deux objets sont égaux s'ils sont accessibles par la même référence.

Toute classe hérite des deux méthodes de `Object`

```
public boolean equals(Object o)
public int hashCode()
```

qui peuvent être redéfinies en respectant the "Object Contract".

### The Object Contract

- `equals` doit définir une relation d'équivalence;
- `equals` doit être consistente. (Plusieurs appels donnent le même résultat);
- `x.equals(null)` doit être faux (si `x` est une référence);
- `hashCode` doit donner la même valeur sur des objets égaux par `equals`.



Une classe peut redéfinir la méthode `equals()`.

```
class Rectangle extends Shape{
  final int width;
  final int height;
  Rectangle(int width, int height) {
    this.width = width;
    this.height = height;
  }
  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Rectangle))
      return false;
    Rectangle rarg = (Rectangle)o;
    return (width == rarg.width)
      && (height == rarg.height);
  }
  @Override
  public int hashCode() {
    return new Integer(width).hashCode() + new Integer(height).hash(
  }
```

Remarquer que l'argument de `equals` est de type `Object`. Si l'argument était `Rectangle`, la méthode serait surchargée. Elle serait alors ignorée lors d'un appel avec un argument de type `Shape` qui référence un `Rectangle`. La comparaison entre les deux rectangles serait alors incorrecte.



Soit la classe `Duration` :

```
public class Duration {
  private final int hours;
  private final int seconds;
  public Duration(int hours, int seconds) {
    this.hours = hours;
    this.seconds = seconds;
  }
  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Duration))
      return false;
    Duration d = (Duration)o;
    return ((hours == d.hours) &&  (seconds == d.seconds));
  }
  @Override
  public int hashCode() {
    return new Integer(hours).hashCode() +new Integer(seconds).hash(
  }
```

On considère la classe dérivée `NanoDuration` :

```
public class NanoDuration extends Duration {
  private final int nanoSeconds;
  public NanoDuration(int hours, int seconds, int nanoSeconds) {
    super(hours,seconds);
    this.nanoSeconds = nanoSeconds;
  }
  @Override
  public boolean equals(Object o) {
    if (!(o instanceof NanoDuration))
      return false;
    NanoDuration nd = (NanoDuration)o;
    return super.equals(nd) && (nanoSeconds == nd.nanoSeconds);
  }
  @Override
  public int hashCode() {
    return super.hashCode()+ new Integer(nanoSeconds).hashCode();
  }
```



Cette définition qui a l'air correcte viole le contrat de symétrie :

```
public class Test{
  public static void main(String[] args) {
    Duration d = new Duration(2,80);
    NanoDuration nd = new NanoDuration(2,80,100);
    System.out.println(d.equals(nd)); //true
    System.out.println(nd.equals(d)); //false
  }
```

En fait, il n'y a pas de solution en redéfinissant uniquement `equals` dans la classe dérivée. Une première méthode consiste à changer `equals` dans la classe mère.

```
public class Duration {
  ...
  @Override
  public boolean equals(Object o) {
    if (o == null)  || (! o.getClass().equals(getClass()))
      return false;
    Duration d = (Duration)o;
    return hours == d.hours &&  seconds == d.seconds;
  }
  ...
```

Quel est l'inconvénient de cette méthode ?
Réponse : on perd la possibilité de comparaisons pour d'éventuelles autres classes dérivées.

Une deuxième méthode consiste à utiliser la délégation (ou composition) au lieu la dérivation.

```
public class NanoDuration {
  final Duration d;
  final int nanoSeconds;
  ...
}
```

Les chaînes de caractères

**La classe `java.lang.String`**

- La classe `String` est `final` (ne peut être dérivée).
- Elle utilise un tableau de caractères (membre privé de la classe).
- Un objet de la classe `String` ne peut être modifié. (On doit créer un nouvel objet).

```
String nom = "toto" + "tata";
System.out.println(nom.length()); // 8
System.out.println(nom.charAt(2)); // t
```

On peut construire un objet `String` à partir d'un tableau de caractères :

```
char table = {'t','o','t','o'};
String s = new String(table);
```

et inversement :

```
char[] table= "toto".toCharArray();
```

Conversion d'un entier en chaîne de caractère :

```
String one = String.valueOf(1); // methode statique
                                   qui appelle toString()
```

et inversement :

```
int i = Integer.valueOf("12"); // ou bien :
int i = Integer.parseInt("12");
```



Comparaison des chaînes de caractères : on peut utiliser la méthode `equals()` :

```
String s = "toto";
String t = "toto";
if (s.equals(t)) ...  // true
```

La méthode `compareTo()` est l'équivalent du `strcmp()` du C.

**Le paquetage java.util.regex**

Ce paquetage permet de faire une analyse lexixale d'un texte, et de rechercher des motifs dans un texte. La classe `java.util.Scanner` utilise ce paquetage.
Un `Pattern` est construit à partir d'une chaîne de caractères représentant une expression régulière (c'est un automate fini).
Un `Matcher` est un objet prêt à parcourir une chaîne passée en argument pour rechercher le motif. La recherche a lieu à l'aide de les méthodes `matches(), lookingAt() et find()`.

Exemple

```
Pattern p = Pattern.compile("a*b"); // Création d'un Pattern
Matcher m = p.matcher("aaaaab");
// scanne la chaîne entière à partir du début
boolean b = m.matches(); // true
m = p.matcher("aaaaabaaaaab");
// scanne la chaîne à partir du début
b = m.lookingAt(); // true
m.reset();
b = m.matches(); // false
// recherche la prochaine occurrence du motif
m.reset();
b = m.find(); //true
b = m.find(); //true
b = m.find(); //false
```



Raccourci :

```
boolean b = Pattern.matches("a*b", "aaaaab"); // true
```

On peut faire afficher l'indice de début du motif dans la chaîne ainsi que l'instance du motif trouvée.

```
public static void main(String[] args){
    Pattern p = Pattern.compile("aa");
    Matcher m = p.matcher("baaaaaaaaaaba");
      while (m.find()){
      System.out.println("indice "+ m.start()+ " "+m.group());
    }
}
```

```
indice 1 aa
indice 3 aa
indice 5 aa
indice 7 aa
indice 9 aa
```

La sous-chaîne la plus longue vérifiant le motif est choisie en cas d'ambiguïté.

```
public static void main(String[] args){
  Pattern p = Pattern.compile("a*");
  Matcher m = p.matcher("baaaaaaaaaaba");
    while(m.find()){
    System.out.println("indice "+m.start()+" "+m.group());
  }
}
```

```
indice 0
indice 1 aaaaaaaaaa
indice 11
indice 12 a
indice 13
```



On peut changer ce comportement par défaut à l'aide de `?`.

```
public static void main(String[] args){
  Pattern p = Pattern.compile("a+?");
  Matcher m = p.matcher("baaaaaaaaaaba");
    while (m.find()){
    System.out.println("indice "+ m.start()+ " "+m.group());
  }
}
```

On peut repérer des sous-expressions dans les expressions.

```
public static void main(String[] args){
 // Groupe 1 Groupe2 Groupe3
  Pattern p = Pattern.compile("(aac(b*))|(aa)");
   Matcher m = p.matcher("baaaaaaaaaabaacbba");
    while (m.find()){
    System.out.print("indice "+m.start()+" "+m.group()+" ");
    for (int i=1; i<= m.groupCount();i++)
      System.out.print("groupe "+ i+": "+m.group(i)+" ");
      System.out.println();
    }
}
```

```
indice 1 aa groupe 1: null groupe 2: null groupe 3: aa
indice 3 aa groupe 1: null groupe 2: null groupe 3: aa
indice 5 aa groupe 1: null groupe 2: null groupe 3: aa
indice 7 aa groupe 1: null groupe 2: null groupe 3: aa
indice 9 aa groupe 1: null groupe 2: null groupe 3: aa
indice 12 aacbb groupe 1: aacbb groupe 2: bb groupe 3: null
```

Les remplacements se font avec `String replaceFirst()` et `String replaceAll()`.

```
public static void main(String[] args){
  Pattern p = Pattern.compile("(xxc(y*))");
  Matcher m = p.matcher("baaaxxcaaaaaaabxxcyya");
  System.out.println(m.replaceAll("toto$2"));
}
```

donne

```
baaatotoaaaaaaabtotoyya
```

```
public static void main(String[] args){
  Pattern p = Pattern.compile("((David)(.*)((Marion))");
  Matcher m = p.matcher("David aime Marion");
  System.out.println(m.replaceAll("$4 $3 $2"));
}
```

donne

```
Marion  aime  David
```

### La class java.util.Scanner

La classe `Scanner` a plusieurs constructeurs dont

- `Scanner(File source)`
- `Scanner(InputStream source)`
- `Scanner(String source)`
- `Scanner(Readable source)`

Utilisation déjà vue :

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```



```
String input = "Vincent mit    500 ânes   \n dans un pré";
Scanner sc = new Scanner(input).useDelimiter("\\s+");
while (sc.hasNext())
  System.out.println(sc.next());
sc.close();
```

donne

```
Vincent
mit
500
ânes
dans
un
pré
```

Le code ci-dessous a le même effet que `sc.nextInt()`.

```
String input = "Vincent mit 500 ânes dans un pré";
Scanner sc = new Scanner(input);
sc.findInLine("(.+?)(\\d+)(.+)");
MatchResult result = sc.match();
System.out.println(result.group(2)); //affiche 500
sc.close()
```



## Outils mathématiques

On peut trouver des outils mathématiques dans les deux classes et le paquetage suivants :

- `java.lang.Math`
- `java.util.Random`
- `java.math` (pour le travail sur des entiers ou flottants longs)

Exemple : `int maximum = Math.max(3,4);`

Exemple : Tirer au hasard un entier entre 100 et 1000 (les deux compris).

`int maximum = 100 + (int)(Math.random()*901);`

- Une instruction arithmétique sur les entiers peut lever l'exception `ArithmeticException` :

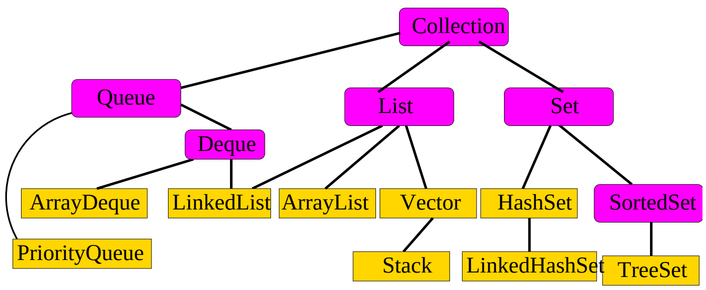
```
try { int i = 1/0;}
catch (ArithmeticException e) {...}
```

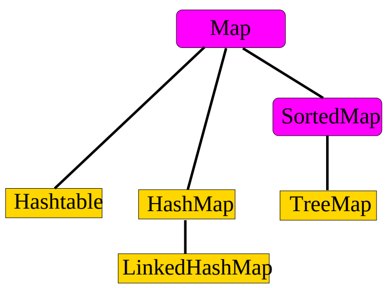
- Une instruction arithmétique sur les flottants (`double`) ne lève pas d'exception. Une expression flottante peut prendre trois valeurs particulières :

| | |
|---|---|
| `POSITIVE_INFINITY` | 1.0/0.0 |
| `NEGATIVE_INFINITY` | −1.0/0.0 |
| `NaN` | 0.0/0.0 `// Not a Number` |



## Ensembles structurés, itérateurs et comparateurs



## Vue d'ensemble

Deux paquetages

- java.util pour les ensembles, collections, itérateurs.
- java.util.concurrent structures supplémentaires pour la programmation concurrente.

Deux interfaces

- Collection<E> pour les ensembles d'objets, avec ou sans répétition.
- Map<K,V> pour les tables, c'est-à-dire des ensembles de couples (clé, valeur), où la clé et la valeur sont de types paramétrés respectifs <K,V>. Chaque clé existe en un seul exemplaire mais plusieurs clés distinctes peuvent être associées à une même valeur.

Des itérateurs sur les collections : ils permettent de parcourir une collection.

- Iterator<E> interface des itérateurs,
- ListIterator<E> itérateur sur les séquences.
- Enumeration<E> ancienne forme des itérateurs.

De plus, deux classes d'utilitaires

- Collections avec des algorithmes de tri etc,
- Arrays algorithmes spécialisés pour les tableaux.



Les opérations principales sur une collection

- add pour ajouter un élément.
- remove pour enlever un élément,
- contains test d'appartenance,
- size pour obtenir le nombre d'éléments,
- isEmpty pour tester si l'ensemble est vide.

Le type des éléments est un type paramétré,<E>.
Sous-interfaces spécialisées de Collection

- List<E> spécifie les *séquences*, avec les méthodes
  - int indexOf(Object o) position de o.
  - E get(int index) retourne l'objet à la position index.
  - E set(int index, E element) remplace l'élément en position index, et retourne l'élément qui y était précédemment.
- Set<E> spécifie les ensembles sans duplication.
- SortedSet<E> sous-interface de Set pour les ensembles ordonnés.
  - E first() retourne le premier objet.
  - E last() retourne le dernier objet.
  - SortedSet<E> subset(E fromElement, E toElement) retourne une référence vers le sous-ensemble des objets $\geq$ fromElementl et $<$ toElement.

Opérations ensemblistes sur les collections

- boolean containsAll(Collection<?> c) pour tester l'inclusion.



- boolean addAll(Collection<? extends E> c) pour la réunion.
- boolean removeAll(Collection<?> c) pour la différence.
- boolean retainAll(Collection<?> c) pour l'intersection.

Les trois dernières méthodes retournent true si elles ont modifié la collection.



## Implémentation d'une collection

Pour les collections

- ArrayList<E> (recommandée, par tableaux), et LinkedList<E> (par listes doublement chaînées) implémentent List<E>.
- Vector<E> est une vieille classe (JDK 1.0) "relookée" qui implémente aussi List<E>. Elle a des méthodes personnelles.
- HashSet<E> (recommandée) implémente Set<E>.
- TreeSet<E> implémente SortedSet<E>.

Le choix de l'implémentation résulte de l'efficacité recherchée : par exemple, l'accès indicé est en temps constant pour les ArrayList<E>, l'insertion entre deux éléments est en temps constant pour les LinkedList<E>.

Discipline d'abstraction:

- les attributs, paramètres, variables locales sont déclarés avec, comme type, une interface (List<Integer>, Set<Double>),
- les classes d'implémentation ne sont utilisées que par leurs constructeurs.

Exemple

```
List<Integer> l = new Arraylist<Integer>();
Set<Integer> s = new HashSet<Integer>();
```

Exemple : Programme qui détecte une répétition dans les chaînes de caractères d'une ligne.

```
import java.util.Set;
import java.util.HashSet;

class SetTest {
  public static void main(String[] args) {
    final Set<String> s = new HashSet<String>();
    for (String w:args)
      if (!s.add(w))
        System.out.println("Déjà vu : " + w);
    System.out.println(s.size() + " distincts : " + s);
  }
}
```

```
$ java SetTest a b c a b d
Déjà vu : a
Déjà vu : b
4 distincts : [d, a, c, b] //toString() de la collection
```



## Itérateurs

L'interface `Iterator<E>` définit les itérateurs.

Un *itérateur* permet de parcourir l'ensemble des éléments d'une collection.

Java 2 propose deux schémas, l'interface `Enumeration<E>` et l'interface `Iterator<E>`.

L'interface `java.util.Iterator` a trois méthodes

- `boolean hasNext()` qui teste si le parcours contient encore des éléments;
- `E next()` qui retourne l'élément suivant, si un tel élément existe (et lève une exception sinon).
- `void remove()` qui supprime le dernier élément retourné par `next`.

Les collections implémentent l'interface `Interface Iterable<T>`, ce qui permet de les parcourir aussi avec la boucle foreach.



```
import java.util.*;
class HashSetTest {
  public static <E> void printAll(Collection<E> c)  {
    for (Iterator<E> i = c.iterator(); i.hasNext(); )
      System.out.println(i.next());
  }
  public static void main(String[] args){
    final Set<Object> s = new HashSet<Object>();
    s.add(new Person("Pierre", 23));
    s.add(new Person("Anne", 20));
    s.add("Université");
    s.add("Marne-la-Vallée");
    printAll(s);
    // copie des références, pas des objets,
    // avec clone() de HashSet.
    final Set<Object> t
    = (Set<Object>) ((HashSet<Object>) s).clone();//unsafe cast
    System.out.println(s.size());
    printAll(t);
    Iterator<Object> i = t.iterator();
      while(i.hasNext())
       if (i.next() instanceof Person) i.remove();
    printAll(t);
}}
```

Avec les résultats

```
$ java HashSetTest
Marne-la-Vallée
Université
Name: Anne, age: 20
Name: Pierre, age: 23
4
Marne-la-Vallée
Université
Name: Anne, age: 20
Name: Pierre, age: 23
Marne-la-Vallée
Université
```



Observer le désordre.

Détails sur les itérateurs.

- la méthode `Iterator<E> iterator()` de la collection positionne l'itérateur au "début",
- la méthode `boolean hasNext()` teste si l'on peut progresser,
- la méthode `E next()` avance d'un pas dans la collection, et retourne l'élément *traversé*.
- la méthode `void remove()` supprime l'élément référencé par `next()`, donc pas de `remove()` sans `next()`.

```
|A B C     iterator(), hasNext() = true
 A|B C     next() = A, hasNext() = true
 A B|C     next() = B, hasNext() = true
 A B C|    next() = C, hasNext() = false
```

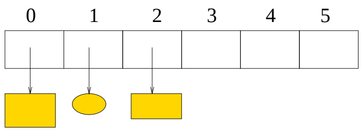
```
Iterator<Character> i = c.iterator();
i.remove(); // NON
i.next();
i.next();
i.remove(); // OK
i.remove(); // NON
```

La classe `java.util.Scanner` implémente `Iterator<String>`.

### Exemple de `ArrayList`

On désire créer un tableau de références sur des objets de type `Shape` qui peuvent être `Rectangle` ou `Ellipse` (déjà vus).

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

class ShapeTest{
  public static void main(String[] args){
    Shape r1 = new Rectangle(6,10);
    Shape r2 = new Rectangle(5,10);
    Shape e = new Ellipse(3,5);
    final List<Shape> liste = new ArrayList<Shape>();
    liste.add(r1);
    liste.add(r2);
    liste.add(1,e);  // on a r1, e, r2
    for (Shape s:liste)
      System.out.println(s.toStringArea());
     // ou bien
    for (Iterator<Shape> it = liste.iterator(); it.hasNext();)
      System.out.println(it.next().toStringArea());
  }
}

$java -classpath ../shape:. ShapeTest
aire = 60.0
aire = 11.780972450961723
aire = 50.0
```






### Itérer sur les listes

Les listes sont des séquences. Un ***iterateur de listes*** implémente l'interface `ListIterator<E>`. Il a des méthodes supplémentaires:

- `E previous()` qui permet de reculer, joint à
- `boolean hasPrevious()` qui retourne vrai s'il y a un élément qui précède.
- `void add(E o)` qui ajoute l'élément juste avant l'itérateur.
- `void set(E o)` qui substitue `o` à l'objet référencé par `next()`

```
import java.util.*;

class LinkedListTest
  public static <E> void printAll(Collection<E> c)  ...
  public static void main(String[] args)
    final List<String> a = new LinkedList<String>();
    a.add("A");
    a.add("B");
    a.add("C");
    printAll(a);  // A B C
    ListIterator<String> i = a.listIterator();
    System.out.println(i.next()); // A | B C  -> A
    System.out.println(i.next()); // A B | C  -> B
    System.out.println(i.hasPrevious()); // true
    System.out.println(i.previous());  // A | B C -> B
    i.add("X");
    printAll(a);  // A X | B C
```



### Comparaison

Java exprime que les objets d'une classe sont comparables, en demandant que la classe implémente l'interface `java.lang.Comparable`.

L'interface `Comparable<T>` déclare une méthode `int compareTo(T o)` telle que `a.compareTo(b)` est

- négatif, si `a < b`.
- nul, si `a = b`.
- positif, si `a > b`.

Une classe `Rectangle` qui implémente cette interface doit définir une méthode `int compareTo(Rectangle o)`. Il est recommandé d'avoir `(a.compareTo(b)==0)` ssi `(a.equals(b))` est vraie.

Exemple : comparaisons de `Person`.

```
import java.util.*;

class Person implements Comparable<Person>{
  protected final String name;
  protected final Integer age;

  public Person(String name, Integer age){
    this.name = name; this.age = age;
  }
  public String getName(){
    return name;
  }
  public Integer getAge(){
    return age;
  }
```



```
  public int compareTo(Person anotherPerson){
    int comp = name.compareTo(anotherPerson.name);
    return (comp !=0) ? comp : age.compareTo(anotherPerson.age);
  }
  public String toString(){return name + " : " + age;}
}

class CompareTest{
  public static void main(String[] args){
    final SortedSet<Person> c = new TreeSet<Person>();
    c.add(new Person("Paul", 21));
    c.add(new Person("Paul", 25));
    c.add(new Person("Anne", 25));
    for (Person p:c)
       System.out.println(p);
  }
}
```

avec le résultat

```
$ java CompareTest
Anne : 25
Paul : 21
Paul : 25
```

## Comparateur

Un *comparateur* est un objet qui permet la comparaison. En Java, l'interface `java.util.Comparator<T>` déclare une méthode `int compare(T o1, T o2)`.

On se sert d'un comparateur

- dans un constructeur d'un ensemble ordonné.
- dans les algorithmes de tri fournis par la classe `Collections`.

Exemple de deux comparateurs de noms :

```
class NameComparator implements Comparator<Person>{
  public int compare(Person o1, Person o2){
    int comp = o1.getName().compareTo(o2.getName());
    if (comp == 0)
      comp = o1.getAge().compareTo(o2.getAge());
    return comp;
  }
}
class AgeComparator implements Comparator<Person>{
  public int compare(Person o1, Person o2){
    int comp = o1.getAge().compareTo(o2.getAge());
    if (comp == 0)
      comp = o1.getName().compareTo(o2.getName());
    return comp;
  }
}
```

Une liste de noms (pour pouvoir trier sans peine).

```
class ComparatorTest{

 public static <E> void printAll(Collection<E> c){
    for (E e = c)
```



```
        System.out.print(e+" ");
      System.out.println();
   }

   public static void main(String[] args){
     final List<Person> c = new ArrayList<Person>();
     c.add(new Person("Paul", 21));
     c.add(new Person("Paul", 25));
     c.add(new Person("Anne", 25));
     printAll(c);
     Collections.sort(c, new NameComparator());
     printAll(c);
     Collections.sort(c, new AgeComparator());
     printAll(c);
   }
 }
```

Et les résultats :

```
Paul : 21 Paul : 25 Anne : 25  // ordre d'insertion
Anne : 25 Paul : 21 Paul : 25  // ordre sur noms
Paul : 21 Anne : 25 Paul : 25  // ordre sur ages
```



## Les tables ou `Map`

L'interface `Map<K,V>` spécifie les tables, des ensembles de couples (clé, valeur). Les clés ne peuvent être dupliquées, au plus une valeur est associée à une clé.

- `V put(K key, V value)` insère l'association `(key, value)` dans la table et retourne la valeur précédemment associée à la clé ou bien `null`.
- `boolean containsKey(Object key)` retourne vrai s'il y a une valeur associée à cette clé.
- `V get(Object key)` retourne la valeur associée à la clé dans la table, ou `null` si `null` était associé ou si `key` n'est pas une clé de la table.
- `V remove(Object key)` supprime l'association de clé `key`. Retourne la valeur précédemment associée. Retourne `null` si `null` était associé ou si `key` n'est pas une clé de la table.

La sous-interface `SortedMap<K,V>` spécifie les tables dont l'ensemble des clés est ***ordonné***.



Implémentation d'une table

Pour les tables

- `HashMap<K,V>` (recommandée), implémente `Map<K,V>`.
- `Hashtable<K,V>` est une vieille classe (JDK 1.0) "relookée" qui implémente aussi `Map<K,V>`. Elle a des méthodes personnelles.
- `TreeMap<K,V>` implémente `SortedMap<K,V>`.

La classe `TreeMap<K,V>` implémente les opérations avec des arbres rouge-noir.

Un `TreeMap<K,V>` stocke ses clés de telle sorte que les opérations suivantes s'exécutent en temps $O(\log(n))$ :

- `boolean containsKey(Object key)`
- `V get(Object key)`
- `V put(K key, V value)`
- `V remove(Object key)`

pourvu que l'on définisse un bon ordre. L'interface `java.util.Comparator` permet de spécifier un comparateur des clés.

## Exemple : formes nommées

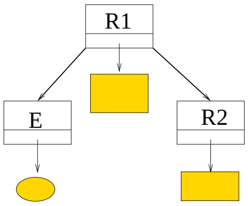
On associe un nom à chaque forme. Le nom est la *clé*, la forme est la *valeur associée* .

L'interface `Shape`, et les classes `Rectangle` et `Ellipse` sont comme d'habitude.

```
import java.util.*;

public class ShapeMapTest{
  public static void main(String[] args){
    Shape r1 = new Rectangle(6,10);
    Shape r2 = new Rectangle(5,10);
    Shape e = new Ellipse(3,5);
    final Map<String,Shape> tree  = new TreeMap<String,Shape>();
    tree.put("R2",r2);
    tree.put("R1",r1);
    tree.put("E",e);
    System.out.println(tree.get("R1").toStringArea());
  }
}
```

On obtient :

```
$ java ShapeMapTest
aire = 60.0
```






Si l'on désire trier les clés en ordre inverse, on change le comparateur.

La classe `java.util.TreeMap<K,V>` possède un constructeur qui permet de changer le comparateur :

`TreeMap(Comparator<? super K> c)`

Le programme devient :

```
import java.util.*;
//ordre inverse
public class OppositeComparator implements Comparator<String>{
  public int compare(String o1, String o2){
    if (o1.compareTo(o2) > 0) return -1;
    if (o1.compareTo(o2) < 0) return 1;
    return 0;
  }
}
```

Cette méthode lève une `NullPointerException` (qui est une `RunTimeException`) si `o1` est `null`. Le reste de la vérification est délégué à `compareTo`.

```
class OppositeTest{
  public static void main(String[] args){
    Shape r1 = new Rectangle(6,10);
    Shape r2 = new Rectangle(5,10);
    Shape e = new Ellipse(3,5);
    Comparator<String> c = new OppositeComparator();
    final SortedMap<String,Shape> tree
                  = new TreeMap<String,Shape>(c);
    tree.put("R2",r2);
    tree.put("R1",r1);
    tree.put("E",e);
    System.out.println(tree.firstKey() + " " + tree.lastKey());
    // affiche R2 E
  }
```



```
}
$javac -classpath ../shape:. OppositeTest.java
$java  -classpath ../shape:. OppositeTest
R2 E
```



## Itérer dans les tables

Les tables n'ont pas d'itérateurs.

Trois méthodes permettent de *voir* une table comme un ensemble

- `Set<K> keySet()` retourne l'ensemble (`Set<K>`) des clés;
- `Collection<V> values()` retourne la collection des valeurs associées aux clés;
- `Set<Map.Entry<K,V>> entrySet()` retourne l'ensemble des couples (clé, valeur). Ils sont de type `Map.Entry<K,V>` qui est une interface statique interne à `Map<K,V>`.

```
Map<String,Shape> m = ...;
Set<String>  keys = m.keySet();
Set<Map.Entry<String,Shape>> pairs = m.entrySet();
Collection<Shape> values= m.values();
```

On peut ensuite itérer sur ces ensembles :

```
for (Iterator<String> i = keys.iterator(); i.hasNext(); )
  System.out.println(i.next());

for (Iterator<Shape> i = values.iterator(); i.hasNext(); )
  System.out.println(i.next());

for (Iterator<Map.Entry<String,Shape>> i = pairs.iterator();
    i.hasNext(); ){
  Map.Entry<String,Shape> e = i.next();
  System.out.println(e.getKey() + " -> " + e.getValue());
}
```

ou utiliser les boucles foreach.

## Exemple : construction d'un index

On part d'une suite d'entrées formées d'un mot et d'un numéro de page, comme

```
22, "Java"
23, "Itérateur"
25, "Java"
25, "Map"
25, "Java"
29, "Java"
```

et on veut obtenir un "index", comme

```
Itérateur [23]
Java [22, 25, 29]
Map [25]
```

Chaque mot apparaît une fois, dans l'ordre alphabétique, et la liste des numéros correspondants et donnée en ordre croisssant, sans répétition.



```
import java.util.*;

class Index {
  private final SortedMap<String,Set<Integer>> map;

  Index() {
    map = new TreeMap<String,Set<Integer>>();
  }
  public void myPut(int page, String word) {
    Set<Integer> numbers = map.get(word);
    if (numbers == null) {
      numbers = new TreeSet<Integer>();
      map.put(word, numbers); // la vraie méthode put
    }
    numbers.add(page);
  }
  public void print(){
    Set<String> keys = map.keySet();
    for (String word: keys)
       System.out.println(word + " " + map.get(word));
  }
}
class IndexTest{
  public static Index makeIndex(){
    Index index = new Index();
    index.myPut(22,"Java");
    index.myPut(23,"Itérateur");
    index.myPut(25,"Java");
    index.myPut(25,"Map");
    index.myPut(25,"Java");
    index.myPut(29,"Java");
    return index;
  }
  public static void main(String[] args){
    Index index = makeIndex();
    index.print();
  }
}
```



## Algorithmes

Les classes `Collections` et `Arrays` (attention au "s" final) fournissent des algorithmes dont la performance et le comportement est garanti. Toutes les méthodes sont statiques.

`Collections`:

- `min`, `max`, dans une collection d'éléments comparables;
- `sort` pour trier des listes (tri par fusion);

```
List<Integer> l;
...
Collections.sort(l);
```

La signature de cette méthode `sort` est (!)

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

- `binarySearch` recherche dichotomique dans les listes ordonnées.
- `copy` copie de listes, par exemple,

```
List<Integer> source = ...;
List<Integer> dest;
Collections.copy(dest, source);
```

- `synchronizedCollection` pour "synchroniser" une collection : elle ne peut être modifiée durant l'exécution d'une méthode.

`Arrays`:

- `binarySearch` pour la recherche dichotomique, dans les tableaux;
- `equals` pour tester l'égalité des contenus de deux tableaux, par exemple



```
int[] a, b ...;
boolean Arrays.equals(a,b);
```

- `sort` pour trier un tableau (quicksort), par exemple

```
int[] a;
...
Arrays.sort(a);
```

Exemple : tirage de loto.

```
import java.util.*;

class Loto {
  public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<Integer>(49);
    for (int i=0; i < 49; i++)
      numbers.add(i);
    Collections.shuffle(numbers); // mélange
    List<Integer> drawing = numbers.subList(0,6);//les 6 premières
    Collections.sort(drawing); // tri
    System.out.println(drawing); // et les voici
  }
}
```

Résultat:

```
> java Loto
[6, 17, 24, 33, 41, 42]
> java Loto
[15, 24, 28, 41, 42, 44]
> java Loto
[27, 30, 35, 42, 44, 46]
```

## "Double ended queue" Deque<E>

Depuis Java 1.6 on peut définir des files FIFO ou LIFO (piles) à l'aide de l'interface `java.util.Deque` qui permet de manipuler une file par les deux bouts.

Les implémentations des `Deque`

- `ArrayDeque<E>`,
- `LinkedList<E>`.

Les opérations principales sur les `Deque` sont

- `addFirst`, `offerFirst` pour ajouter un élément en tête,
- `addLast`, `offerLast` pour ajouter un élément à la fin,
- `removeFirst`, `pollFirst` pour enlever un élément en tête,
- `removeLast`, `pollLast` pour enlever un élément à la fin,
- `gerFirst`, `peekFirst` pour regarder un élément en tête,
- `getLast`, `peekLast` pour regarder un élément à la fin.

Chaque méthode (sur chaque ligne ci-dessus) a deux formes : la première renvoie une exception (Runtime) si l'opération échoue. La deuxième renvoie null ou false.
L'interface donne un itérateur en sens inverse.

- `Iterator<E> descendingIterator()`



### Exemple de file FIFO

```
import java.util.*;

public class Test {
  public static void main(String[] args) {
      final Deque<Integer> deque = new LinkedList<Integer>();
      deque.addLast(10);
      deque.addLast(20);
      deque.addLast(30);
      for (Integer i:deque){
         System.out.print(i+ " ");
      }
      System.out.println();
      for (Iterator<Integer> it = deque.descendingIterator();
        it.hasNext();){
         System.out.print(it.next()+ " ");
      }
      System.out.println();
      System.out.println(deque.removeFirst());
      System.out.println(deque);
      System.out.println(deque.removeFirst());
      System.out.println(deque.removeFirst());
      System.out.println(deque.pollFirst());
      // Exception java.util.NoSuchElementException
      System.out.println(deque.removeFirst());
  }
}
```

donne

```
10 20 30
30 20 10
10
[20, 30]
20
30
null
Exception in thread "main" java.util.NoSuchElementException
```



## Énumérations

Java 1.5 permet de définir des types énumérés. La classe de base est `java.lang.Enum`.

Exemple d'usage simple :

```
class Test {
    // Il s'agit d'une déclaration de classe
    private enum Color { BLUE, RED, GREEN }

    public static void main(String[] args) {
       Color c = Color.BLUE;
       System.out.println(c);
    }
}
$ javac Test.java
$ ls
Test.class Test$Color.class Test.java
$ java Test
BLUE
```

Une classe interne, qui dérive de la classe abstraite `Enum`, est créée. Les valeurs énumérées sont des champs publics statiques de cette classe.
Un type énuméré peut implémenter des interfaces. Les champs peuvent être utilisés dans les collections, être des clés des tables.



Chaque type énuméré a une méthode `static values()` renvoyant un tableau contenant les valeurs énumérées.

Exemple plus fourni :

```
import java.util.*;

public class CoinTest {
  public enum Coin {
     DOLLAR(1), EURO(10), PESOS(50);

     private final int value;
     //le constructeur n'est pas appelé directement
     Coin(int value) { this.value = value; }
     public int getValue() { return value; }
 }
  public static void main(String[] args) {
     public static void main(String[] args) {
     for (Coin c : Coin.values()) {
        System.out.println(c + ":   "+ c.getValue());
     }
  }
}
$ java CoinTest
DOLLAR:         1
EURO:           10
PESOS:          50
```

## Programmation des listes chaînées suite

On reprend l'exemple d'implémentation des listes simplement chaînées (t.90) contenant des éléments de type `Object` (sans utiliser les collections des API). On écrit son propre itérateur pour ces listes. Dans une première version, on utilise une classe interne nommée. On écrit ensuite une classe interne anonyme.

```
import java.util.*;

public class Liste<E> implements Iterable<E> {
   Cell<E> init;
   public Liste() { init = new ConcreteNil<E>();}
   public int length() { return init.length();}
    // inner class
   class ListeIterator<E> implements Iterator<E> {
       Cell<E> c;
       ListeIterator() {
         c = init;
       }
       public boolean hasNext() {
         return (c.length() != 0);
       }
       public E next() {
          if (! hasNext()) {
             throw new NoSuchElementException();
          }
          E o = ((Cons<E>)c).getElem();
          c = ((Cons<E>)c).getNext();
          return o;
       }
       public void remove() {
          throw new UnsupportedOperationException();
       }
   }
```



```
      public Iterator<E> iterator() {
         return new ListeIterator();
      }
   }

public class ListeTest {
  public static void main(String[] args) {
    Liste<Integer> l = new Liste<Integer>();
    l.init = new ConcreteCons<Integer>
             (1, new ConcreteCons<Integer>
             (2,new ConcreteNil<Integer>()));
    System.out.println("liste de longueur "+ l.length());
    for (Integer o : l)
        System.out.println(o);
  }
}
```

L'exécution donne :

```
liste de longueur 2
1
2
```

La compilation crée un fichier `Liste$ListeIterator.class` pour la classe interne.

```
import java.util.*;

public class Liste implements Iterable<E> {
   Cell<E> init;
   public Liste() { init = new ConcreteNil<E>();}
   public int length() { return init.length();}

    public Iterator<E> iterator() {
       return new Iterator<E>() {
           Cell<E> c = init;
           public boolean hasNext() {
             return (c.length() != 0);
           }
           public E next() {
```



```
             if (hasNext()) {
               E o = ((Cons<E>)c).getElem();
               c = ((Cons<E>)c).getNext();
               return o;
           }
             else throw new NoSuchElementException();
           }
           public void remove() {
             throw new UnsupportedOperationException();
           }
       };
    }
}
```

La compilation crée un fichier `Liste$1.class` pour la classe interne anonyme.

Exercice : écrire le `remove()`.



## Classes internes

- Une classe interne est une classe déclarée à l'intérieur d'une autre classe.
- Une classe interne ne contient pas de membre statiques. Une classe interne non statique (inner class comme ci-dessus) a accès à tous les membres de sa classe englobante. La classe englobante a accés à tous les membres de sa classe interne.
- On peut obtenir une référence sur l'instance de la classe englobante `Toto` par `Toto.this`.
- Une classe interne (nommée ou anonyme) peut être définie à l'intérieur d'une méthode (implémentation 2 ci-dessus). Ceci est utilisé en interface graphique, ou pour écrire des itérateurs.
- La machine virtuelle java ne connaît pas les classes internes (comme elle ne connaît pas les générics).

# 4

## Les flots

1. Généralités
2. Flots d'octets, flots de caractères
3. Les filtres
4. Comment lire un entier
5. Manipulation de fichiers
6. Flots d'objets ou sérialisation



### Généralités

Un *flot* (*stream*) est un canal de communication dans lequel on peut lire ou écrire. On accède aux données séquentiellement.

Les flots prennent des données, les transforment éventuellement, et sortent les données transformées.

*Pipeline ou filtrage*

Les données d'un flot d'entrées sont prises dans une *source*, comme l'entrée standard ou un fichier, ou une chaîne ou un tableau de caractères, ou dans la sortie d'un autre flot d'entrée.

De même, les données d'un flot de sortie sont mises dans un *puit*, comme la sortie standard ou un fichier, ou sont transmises comme entrées dans un autre flot de sortie.

En Java, les flots manipulent soit des octets, soit des caractères. Certains manipulent des données typées.

Les classes sont toutes dans le paquetage `java.io` (voir aussi `java.net`, par exemple la classe `java.net.URL`).
Les classes de base sont

```
File
RandomAccessFile

InputStream
OutputStream

Reader
Writer

StreamTokenizer
```

Les `Stream`, `Reader` et `Writer` sont abstraites.



Les `Stream` manipulent des octets, les `Reader` et `Writer` manipulent des caractères.

Il exite aussi des classes `StringReader` et `StringWriter` pour manipuler les chaînes comme des flots.

### Hiérarchie des classes

Fichiers

```
File
FileDescriptor
RandomAccessFile
```

Streams

```
InputStream
   ByteArrayInputStream
   FileInputStream
   FilterInputStream
      BufferedInputStream
      DataInputStream
      LineNumberInputStream
      PushbackInputStream
   ObjectInputStream
   PipedInputStream
   SequenceInputStream

OutputStream
   ByteArrayOutputStream
   FileOutputStream
   FilterOutputStream
      BufferedOutputStream
      DataOutputStream
      PrintStream
   ObjectOutputStream
   PipedOutputStream
```



Reader Writer

```
Reader                        Writer
   BufferedReader                BufferedWriter
      LineNumberReader           CharArrayWriter
   CharArrayReader
   FilterReader                  FilterWriter
      PushbackReader
   InputStreamReader             OutputStreamWriter
      FileReader                    FileWriter
   PipedReader                   PipedWriter
   StringReader                  StringWriter
                                 PrintWriter
```

## Flots d'octets et de caractères

*Les flots d'octets en lecture*

Objet d'une classe dérivant de `InputStream`.
`System.in` est un flot d'octets en lecture.
Méthodes pour lire *à partir* du flot :

- `int read()` : lit un octet dans le flot, le renvoie comme octet de poids faible d'un `int` ou renvoie $-1$ si la fin du flot est atteinte;
- `int read(byte[] b)` : lit au plus `b.length` octets dans le flot et les met dans `b`;
- `int read(byte[] b, int off, int len)` : lit au plus `len` octets dans le flot et les met dans `b` à partir de `off`;
- `int available()` : retourne le nombre d'octets dans le flot;
- `void close()` : ferme le flot.

*Les flots d'octets en écriture*

Objet d'une classe dérivant de `OutputStream`.
`System.out` est de la classe `PrintStream`, qui dérive de `FilterOutputStream` qui dérive de `OutputStream`.
Méthodes pour écrire *dans* le flot:

- `void write(int b)` : écrit dans le flot l'octet de poids faible de `b`;
- `void write(byte[] b)` : écrit dans le flot tout le tableau;



- `int write(byte[] b, int off, int len)` : écrit dans le flot `len` octets à partir de `off`;
- `void close()` : ferme le flot.



### Lire un octet

```
import java.io.*;

public class ReadTest {
  public static void main(String[] args){
    try {
      int i = System.in.read();
      System.out.println(i);
    } catch (IOException e) {};
  }
}
```

On obtient :

```
$ java ReadTest
a
97
```

### Lire des octets

```
public class ReadTest {
  static int EOF = (int) '\n';
  public static void main(String[] args) throws IOException {
    int i;
    while ((i = System.in.read()) != EOF)
      System.out.print(i + " ");
    System.out.println("\nFin");
  }
}
```

On obtient :

```
$ java ReadTest
béal
98 233 97 108
Fin
```

*Les flots de caractères en lecture*

Objet d'une classe dérivant de `Reader`.
Méthodes pour lire *à partir* du flot :



- `int read()` : lit un caractère dans le flot, le renvoie comme octets de poids faible d'un `int` ou renvoie $-1$ si la fin du flot est atteinte;
- `int read(char[] b)` : lit au plus `b.length` caractères dans le flot et les met dans `b`;
- `int read(char[] b, int off, int len)` : lit au plus `len` caractères dans le flot et les met dans `b` à partir de `off`;
- `int available()` : retourne le nombre d'octets dans le flot;
- `void close()` : ferme le flot.

*Les flots d'octets en écriture*

Objet d'une classe dérivant de `Writer`.
Les méthodes sont analogues à celles des flots d'octets.

## Les filtres

Un *filtre* est un flot qui *enveloppe* un autre flot.

Les données sont en fait lues (ou écrites) dans le flot enveloppé après un traitement (codage, bufferisation, etc). Le flot enveloppé est passé en argument du constructeur du flot enveloppant.

Les filtres héritent des classes abstraites :

- `FilterInputStream` (ou `FilterReader`);
- `FilterOutputStream` (ou `FilterWriter`).

Filtres prédéfinis :

- `DataInputStream`, `DataOutputStream` : les méthodes sont `write`*Type*`()`, `read`*Type*`()`, où *Type* est `Int, Char, Double, ...`;
- `BufferedInputStream` : permet de bufferiser un flot;
- `PushBackInputStream`: permet de replacer des données lues dans le flot avec la méthode `unread()`;
- `PrintStream` : `System.out` est de la classe `PrintStream`.
- `InputStreamReader` : transforme un `Stream` en `Reader`;
- `BufferedReader` : bufférise un flot de caractères;
- `LineNumberReader` : pour une lecture de caractères ligne par ligne;



## Lire des entiers

### Un entier avec BufferedReader

```
class Read{
  public static int intRead() throws IOException{
    InputStreamReader in = new InputStreamReader(System.in);
    BufferedReader data = new BufferedReader(in);
    String s = data.readLine();
    return Integer.parseInt(s);
  }
}

class ReadTest{
  public static void main(String[] args) throws IOException{
    int i = Read.intRead();
    System.out.println(i);
  }
}
```

La méthode `String readLine()` de la classe `BufferedReader` retourne la ligne suivante. La classe `LineNumberReader` dérive de la classe `BufferedReader`.



### Lire un texte sur l'entrée standard

```
class Test {
  public static void main(String[] args) throws IOException {
    BufferedReader in = null;
    try {
        in = new BufferedReader(
             new InputStreamReader(System.in));
        String s;
        while ((s = in.readLine()) != null) {
            System.out.print("> ");
            s = s.toUpperCase();
            System.out.println(s);
        }
    } catch (IOException e) {
    } finally {
        if (in != null) in.close();
    }
   }
}
```

```
marie
> MARIE
pierre
> PIERRE
béal
> BÉAL
```



### Lire une suite d'entiers avec StreamTokenizer

Un **StreamTokenizer** prend en argument un flot (reader) et le fractionne en "token" (lexèmes). Les attributs sont

- `nval` contient la valeur si le lexème courant est un nombre (double)
- `sval` contient la valeur si le lexème courant est un mot.
- `TT_EOF, TT_EOL, TT_NUMBER, TT_WORD` valeurs de l'attribut `ttype`. Si un token n'est ni un mot, ni un nombre, contient l'entier représentant le caractère.

```
class MultiRead {
  public static void read() throws IOException{
    StreamTokenizer in;
    InputStreamReader w = new InputStreamReader(System.in);
    in = new StreamTokenizer(new BufferedReader(w));
    in.quoteChar('/');
    in.wordChars('@','@');
    do {
      in.nextToken();
      if (in.ttype == (int) '/')
        System.out.println(in.sval);
      if (in.ttype == StreamTokenizer.TT_NUMBER)
         System.out.println((int) in.nval); // normalement double
      if (in.ttype == StreamTokenizer.TT_WORD)
        System.out.println(in.sval);
    } while (in.ttype != StreamTokenizer.TT_EOF);
  }
}

class MultiReadTest{
  public static void main(String[] args) throws IOException{
    MultiRead.read();
  }
}
```

```
$ cat Paul
0 @I1@ INDI
1 NAME Paul /Le Guen/
0 TRLR

$ java MultiReadTest < Paul
0
@I1@
INDI
1
NAME
Paul
Le Guen
0
TRLR
```



## Manipulation de fichiers

Les *sources* et *puits* des stream et reader sont

- les entrées et sorties standard (`printf`)
- les String (`sprintf`)
- les fichiers (`fprintf`)

Pour les `String`, il y a les `StringReader` et `StringWriter`. Pour les fichiers, il y a les stream et reader correspondants.

- La classe `java.io.File` permet de manipuler le système de fichiers;
- Les classes `FileInputStream` (et `FileOutputStream`) définissent des flots de lecture et d'écriture de fichiers d'octets, et les classes `FileReader` (et `FileWriter`) les flots de lecture et d'écriture de fichiers de caractères.



La classe `File` décrit une représentation d'un fichier.

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

public class FileInformation{
  public static void main(String[] args) throws Exception{
    info(args[0]);
  }

  public static void info(String nom)
      throws FileNotFoundException {
    File f = new File(nom);
    if (!f.exists())
      throw new FileNotFoundException();
    System.out.println(f.getName());
    System.out.println(f.isDirectory());
    System.out.println(f.canRead());
    System.out.println(f.canWrite());
    System.out.println(f.length());
    try {
    System.out.println(f.getCanonicalPath());
    System.out.println(f.toURL());
    } catch (IOException e) {
    }
  }
}
```

On obtient :

```
monge : > ls -l FileInformation.java
-rw-r--r-- 1 beal  beal 638 fév 13 16:08 FileInformation.java
monge : > java FileInformation FileInformation.java
FileInformation.java
false
true
true
638
/home/beal/Java/Programmes5/file/FileInformation.java
file:/home/beal/Java/Programmes5/file/FileInformation.java
```



### Lecture d'un fichier

Un lecteur est le plus souvent défini par

`FileReader f = new FileReader(nom);`

où `nom` est le nom du fichier. La lecture se fait par les méthodes de la classe `InputStreamReader`.
Lecture par blocs.

```
FileInputStream in = new FileInputStream(nomIn);
FileOutputStream out = new FileInputStream(nomOut);
int readLength;
byte[] block = new byte[8192];
while ((readLength = in.read(block)) != -1)
  out.write(block, 0, readLength);
```

Lecture d'un fichier de texte, ligne par ligne.

```
import java.io.*;
class ReadFile {
 public static String read(String f) throws IOException {
  FileReader fileIn = null;
  StringBuilder s = new StringBuilder();
  try {
  fileIn = new FileReader(f);
  BufferedReader in = new BufferedReader(fileIn);
  String line;
  while ((line = in.readLine()) != null)
    s.append(line + "\n");
  } catch (IOException e) {
  } finally {
  if (fileIn != null) fileIn.close();
  return s.toString();
  }
 }
 public static void main(String[] args) throws IOException {
     System.out.println(ReadFile.read("toto"));
 }
}
```

## Les flots d'objets ou sérialisation

- Un *flot d'objets* permet d'écrire ou de lire des objets Java dans un flot.
- On utilise pour cela les filtres `ObjectInputStream` et `ObjectOutputStream`. Ce service est appelé *sérialisation*.
- Les applications qui échangent des objets via le réseau utilisent la sérialisation.
- Pour sérialiser un objet, on utilise la méthode d'un flot implémentant l'interface `ObjectOutput` : `void writeObject(Object o)`.
- Pour désérialiser un objet, on utilise la méthode d'un flot implémentant l'interface `ObjectInput` : `Object readObject()`.
- Pour qu'un objet puisse être inséré dans un flot, sa classe doit implémenter l'interface `Serializable`. Cette interface ne contient pas de méthode.

La première fois qu'un objet est sauvé, tous les objets qui peuvent être atteints à partir de cet objet sont aussi sauvés. En plus de l'objet, le flot sauvegarde un objet appelé *handle* qui représente une référence locale de l'objet dans le flot. Une nouvelle sauvegarde entraîne la sauvegarde du handle à la place de l'objet.



### Exemple de sauvegarde

```
import java.io.*;

public class Pixel implements Serializable {
  private int x, y;
  public Pixel(int x,int y){ this.x = x; this.y = y; }
  public String toString(){ return "(" + x + "," + y + ")"; }

  public void savePixel(String name) throws Exception {
    File f = new File(name);
    ObjectOutputStream out;
    out = new ObjectOutputStream(new FileOutputStream(f));
    out.writeObject(this);
    out.close();
    // fin de la partie sauvegarde

    ObjectInputStream in;
    in = new ObjectInputStream(new FileInputStream(f));
    Pixel oBis = (Pixel) in.readObject();
    in.close();
    System.out.println(this);
    System.out.println(oBis);
    System.out.println(this.equals(oBis));
  }

  public static void main(String[] args) throws Exception{
    Pixel o = new Pixel(1,2);
    o.savePixel(args[0]);
  }
}
```

On obtient :

```
monge :> java Pixel toto
(1,2)
(1,2)
false
```



### Redéfinir l'objet de sauvegarde

Au moment de la sauvegarde, il est possible de remplacer un objet par un autre.

- On définit pour cela la méthode `Object writeReplace()` dans la classe de *l'objet à remplacer*.
- Au moment de la désérialisation, on utilise la méthode `Object readResolve()` de la classe de *l'objet remplacé* pour retourner un objet compatible avec l'original.

Dans l'exemple suivant, une liste d'entiers est remplacée, au moment de son écriture dans un fichier, par un objet de la classe `ListeString` qui contient la liste des entiers sous forme de chaîne de caractères.



```
class Serial {
  public static void main(String[] args) throws Exception{
    Liste l = new Liste(1, new Liste(2,null));
    l.writeListe(args[0]);
    l.readListe(args[0]);
  }
}

class Liste implements Serializable{
  int val;
  Liste next;
  public Liste(int val, Liste next){
    this.val = val ;
    this.next = next;
  }
  Object writeReplace() throws ObjectStreamException{
    Liste tmp;
    StringBuilder buffer = new StringBuilder();
    for (tmp = this; tmp != null; tmp = tmp.next)
      buffer.append(" " + tmp.val);
    return new ListeString(buffer.toString());
  }
  public String toString(){
    return "(" + val + "," + next + ")";
  }
  public void writeListe(String nom) throws Exception {
    ObjectOutputStream out;
    out = new ObjectOutputStream(new FileOutputStream(nom));
    out.writeObject(this);
    out.close();
  }
  public void readListe(String nom) throws Exception {
    ObjectInputStream in;
    in = new ObjectInputStream(new FileInputStream(nom));
    Liste l = (Liste) in.readObject();
    System.out.println(l);
  }
}
```

```
class ListeString implements Serializable {
  String s;
  ListeString(String s) { this.s = s; }

  Object readResolve() throws ObjectStreamException {
    StringTokenizer st = new StringTokenizer(s);
    return resolve (st);
  }
  Liste resolve(StringTokenizer st) {
    if (st.hasMoreTokens()) {
      int val = Integer.parseInt(st.nextToken());
      return new Liste(val, resolve(st));
    }
    return null;
  }
}
```

On obtient :

```
monge :> java Serial toto
(1,(2,null))
monge :> file toto
toto: Java serialization data, version 5
```



# 5

## Introduction à JDBC





## Qu'est-ce que JDBC ?

***Java DataBase Connectivity***

- est une API Java (`java.sql`) permettant de se connecter avec des bases de données relationnelles (SGBDR),
- elle fournit un ensemble de classes et d'interfaces permettant l'utilisation d'un ou plusieurs SGBDR à partir d'un programme Java.
- elle supporte le standard SQL-3, permet la connexion à une ou plusieurs bases, le lancement de requêtes SQL et le traitement des résultats.

***Avantages et inconvénients***

- Java est un excellent candidat pour le développement d'applications de bases de données.
- JDBC permet au programmeur d'écrire un code indépendant de la base de données cible et du moyen de connectivité utilisé.
- Les bases de données relationnelles sont très répandues mais Java est orienté objet. L'idéal serait de traiter des bases de données elle-même orientées objet.
- La compilation ne permet pas de vérifier le code SQL avant l'exécution (voir SQLJ, SQL embarqué dans Java). Il n'y a pas de vérifications de types vis-à-vis de la base de données.



## Principe de fonctionnement

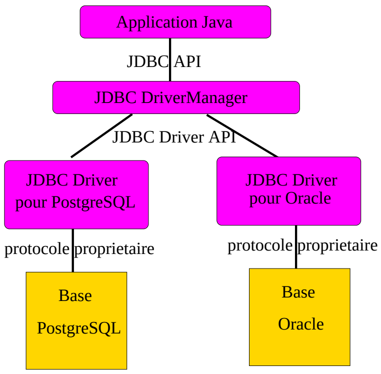
***Connexion avec un DriverManager (ancienne méthode)***

- Chaque base de données utilise un pilote (*driver*) qui lui est propre. Ce pilote permet de convertir les requêtes JDBC dans le langage natif du système de gestion de la base de données relationnelles.
- Ces pilotes (dits drivers JDBC) sont un ensemble de classes et interfaces. Ils sont fournis par les différents constructeurs ou propriétaires (Oracle, PostgreSQL, MySQL, ...).
- On distingue 4 types de drivers (I,II,III,IV) suivant qu'ils contiennent ou non des fonctions natives (non Java) de l'API du SGBDR et suivant le protocole réseau utilisé.
- Par exemple un driver de type IV est écrit en pur Java et communique avec la base avec le protocole réseau de la base. Une applette peut être exécutée si le serveur SGBDR est installé au même endroit que le serveur Web.

## Architecture

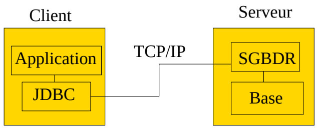
### *Architecture logicielle*

Pour les drivers de type IV, on peut distinguer deux couches.




### *Architecture réseau*

Un modèle client-serveur simple :






### *Connexion avec un objet DataSource*

Les constructeurs de bases de données fournissent les drivers et les implémentations des `DataSource`. Un objet DataSource représente une base de données physique et chaque connexion créée est une connexion à cette base. L'interface `DataSource` possède des méthodes de création de connexions. Un objet data source a des "properties" (propriétés persistantes) contenant des informations comme la localisation du serveur de la base de données, le nom de la base, le protocole réseau utilisé pour communiquer avec le serveur, etc .

Les avantages de ce mode de connexion par rapport à celui via les driver managers sont

- Il n'est pas nécessaire de coder en dur la classe driver;
- Si des propriétés de la base changent, le code des applications utilisateurs n'a pas à être changé;
- Des connexions distribuées et aussi des connexions "stockées" (pooled connexions). Dans ce cas, une connexion refermée peut être recyclée, ce qui réduit les nombre de créations de connexions. L'ouverture d'une connexion est effet coûteuse. Ceci s'opère de façon transparente pour l'utilisateur;
- Un nom logique de la base a été enregistré par l'administrateur système via l'API JNDI (Java Naming and Directory Interface). L'objet DataSource est recherché via cette API. Si la base source est mise sur autre serveur, ceci est transparent à l'utilisateur.



## Utilisation de JDBC

- Importer les paquetages `java.sql`, `javax.sql`, `javax.naming`, (et `javax.sql.rowset` Java version 1.5);
- Se connecter à la base de données;
- Préparer une requête;
- Exécuter la requête;
- Récupérer les données retournées et les traiter;
- Fermer un certain nombre d'objets ouverts au cours des opérations précédentes (dont la connexion à la base).



## Connexion (ancienne méthode)

### *Enregistrement d'un driver*

- On charge une classe `Driver` qui crée une instance d'elle même et s'enregistre auprès du `DriverManager`.

```
Class.forName("org.postgresql.Driver");
```

### *Connexion à la base par un DriverManager*

- Elle se fait via la méthode `getConnection()` de la classe `DriverManager`.
- Cette méthode demande une URL, un nom d'utilisateur et éventuellement un mot de passe (tous de type `String`).
- L'URL n'est pas très normalisée. Elle indique qu'on utilise JDBC, le type du driver ou le type du SGBDR, l'identification de la base distante.
- Exemples d'URLs :

```
String url = "jdbc:postgresql:beal_base";
String url = "jdbc:postgresql://localhost:5432/beal_base";
String url = "jdbc:postgresql://sqletud.univ-mlv.fr:5432/beal_base";
```

- Exemple de connexion. (L'administrateur de la base de nom beal_base a enregistré l'utilisateur beal en l'autorisant à consulter et modifier la base sans contrôle de mot de passe. L'utilisateur beal possède de plus le droit CREATEDB).

```
Class.forName("org.postgresql.Driver");
String url = "jdbc:postgresql:beal_base";
Connection co = DriverManager.getConnection( url, "beal", "");
```

Exemple de connexion-déconnexion (ancienne méthode).

```
public class Creation{

  public Creation() throws Exception {

     Class.forName("org.postgresql.Driver");
     String url = "jdbc:postgresql:beal_base";
     // Autre possibilite pour l'URL
     // String url = "jdbc:postgresql://localhost:5432/beal_base";
     Connection co = DriverManager.getConnection( url, "beal", "");
     if (co != null) co.close();
  }
  public static void main(String[] args) throws Exception{
       new Creation();
  }
}
```

### *Connexion à la base par une DataSource*

```
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("jdbc/beal_base");
Connection co = ds.getConnection("beal","");
```

Une "pooled" connexion sera créée automatiquement si cette implémentation existe. Sinon une connexion standard est créée.



Exemple de connexion-déconnexion avec une `DataSource`.

```
import java.sql.*;
import javax.sql.*;
public class DataSourceConnexion {

  public DataSourceConnexion() {

  org.postgresql.ds.PGPoolingDataSource source =
             new org.postgresql.ds.PGPoolingDataSource();
  source.setDataSourceName("jdbc/beal_base");
  source.setServerName("localhost:5432");
  source.setDatabaseName("beal_base");
  source.setUser("beal");
  source.setPassword("");
  source.setMaxConnections(10);

  Connection co = null;
  try {
    co = source.getConnection();
    // use connection
  } catch (SQLException e) {
    // log error
  } finally {
    if (co != null) {
        try { co.close(); } catch (SQLException e) {}
        }
  }
  }

   public static void main(String[] args) throws Exception{
       new DataSourceConnexion();
   }
}
```



Exemple de connexion-déconnexion avec une `DataSource` et la JNDI.

```
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
import java.io.*;
import org.postgresql.ds.*; // contient PGPoolingDataSource

public class DataSourceAdmin {
  public static void initialize() throws Exception {
      // classe du vendeur
      PGPoolingDataSource source = new PGPoolingDataSource();
      source.setDataSourceName("jdbc/beal_base");
      source.setServerName("localhost:5432");
      source.setDatabaseName("beal_base");
      source.setUser("beal");
      source.setPassword("");
      source.setMaxConnections(10);
      Context ctx = new InitialContext();
      ctx.bind("jdbc/beal_base", source);
  }
  public static void main(String[] args) throws Exception{
       DataSourceAdmin.initialize();
  }
}

public class DataSourceConnexion {
  public DataSourceConnexion() throws Exception {
     // peut lever javax.naming.NamingException
     Context ctx = new InitialContext();
     // peut lever javax.naming.NamingException
     DataSource ds = (DataSource) ctx.lookup("jdbc/beal_base");
     Connection co = null;
     try {
     co = ds.getConnection("beal","");
     SQLWarning w = co.getWarnings();
     if (w != null) System.out.println (w.getMessage());;
```



```
     }
     catch (SQLException e){
     System.out.println (e.getMessage());
     }
     catch (Exception e){
     e.printStackTrace ();
     }
     finally {
     if (co != null) co.close();
     }
  }
   public static void main(String[] args) throws Exception{
       new DataSourceConnexion();
   }
}
```

## java.sql

comporte 18 interfaces dont les suivantes

- `Driver`,
- `Connection`,
- `Statement`, `PreparedStatement`, `CallableStatement`,
- `ResultSet`,
- `ResultSetMetaData`, `DataBaseMetaData`,
- `Blob`, `Clob`, `Array` (depuis SQL-3, Binary Large Object, Character Large Object). Ces interfaces sont destinées à traiter des bases de données où l'on stocke par exemple des images. L'interface `Array` est utilisée pour récupérer un attribut (une seule colonne de la BDR) qui contient plusieurs informations rangées dans un tableau (exemple : les notes d'un étudiant).



## javax.sql

comporte les interfaces suivantes

- `DataSource`,
- `ConnectionPoolDataSource`,
- `PooledConnection`,
- `RowSet`, dérive de `ResultSet`,
- `JDBCRowSet`, `CachedRowSet`, ..., dérivent de `RowSet`.
- `RowSetMetaData`, dérive de `ResultSetMetadata`.



## Manipulations de base

### *Préparer une requête*

- À partir de l'objet de la classe `Connection`, on récupère un objet de la classe `Statement`.

```
Statement st = co.createStatement();
```

- Il existe trois types de `Statement`
  - `Statement` : requêtes statiques simples
  - `PreparedStatement` : requêtes dynamiques pré-compilées avec paramètres.
  - `CallableStatement` : requêtes stockées.

### *Lancer une requête*

- Les méthodes `executeQuery` et `executeUpdate` de la classe `Statement` permettent d'exécuter une requête SQL passée en argument sous la forme d'une `String`.
- La méthode `executeQuery` est utilisée pour les requêtes de type SELECT et renvoie un objet `ResultSet` qui représente les t-uples résultats de la requête.
- La méthode `executeUpdate` est utilisée pour les requêtes modifiant une table (INSERT, UPDATE, CREATE TABLE, DROP TABLE, DELETE). Elle retourne un entier indiquant le nombre de t-uples traités.
- La méthode `execute` exécute n'importe quelle requête.
- Exemple : pour obtenir tous les t-uples de la table Apprenti.



```
Statement st = co.createStatement();
String s = "SELECT * FROM Apprenti;";
ResultSet rs = st.executeQuery(s);
```

- Le code SQL n'est pas interprété par Java mais par le pilote associé lors de la connexion.

### *Récupérer les données retournées et les traiter*

- L'objet de la classe `ResultSet` représente les t-uples résultats. On peut les parcourir avec `next()` et `previous()`.
- Les colonnes sont référencées par leur nom (l'attribut de la relation représentée par la table) ou leur numéro.
- L'accès aux colonnes se fait par une méthode `getString()`, `getInt()`,..., (avec argument `String` ou `int`), suivant le type de l'attribut.
- Les méthodes `Object getObject(String)` ou `Object getObject(int)` renvoient un objet correspondant au type de l'attribut (possibilité de récupérer un objet Java si la base le permet (SQL type JAVA_OBJECT)).

```
...
Statement st = co.createStatement();
String s = "SELECT * FROM Apprenti;";
ResultSet rs = st.executeQuery(s);
String n,p; int i;
while (rs.next()) {
   n = rs.getString("Nom");
   p = rs.getString("Prenom");
   i = rs.getInt("NSS");
   System.out.println(n+" "+p" "+i);
}
rs.close();
st.close();
co.close();
```



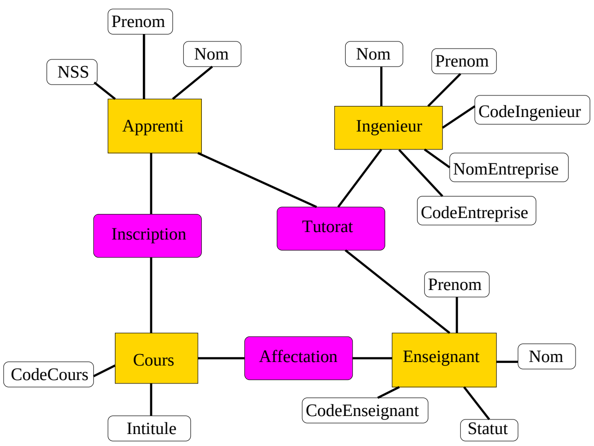
## Correspondances de types

| Type SQL | classe ou interface Java |
|---|---|
| CHAR, VARCHAR | `String` |
| BIT | `boolean` |
| INTEGER | `int` |
| BIGINT | `long` |
| REAL | `float` |
| DOUBLE, FLOAT | `double` |
| DATE | `java.sql.Date` |
| BLOB | `java.sql.Blob` |
| CLOB | `java.sql.Clob` |
| ARRAY | `java.sql.Array` |
| DATE | `java.sql.Date` |
| REF | `java.sql.Ref` |
| STRUCT | `java.sql.Struct` |
| SQL types | `java.sql.Types` |
| ... | ... |

- On peut tester si la valeur d'une colonne est la valeur NULL de SQL avec la méthode `wasNull()` de `ResultSet` qui renvoie vraie si l'on vient de lire une valeur nulle.
- Les méthodes `getInt()` ... convertissent une valeur NULL en une valeur compatible avec le type renvoyé (0 pour les valeurs numériques de base, null pour une référence, false pour un booléen).



## Exemple : du modèle objet au modèle relationnel






## Exemple : création de tables et interrogation

On considère le schéma relationnel formé des relations suivantes.

- Apprenti (Nom, Prenom, NSS)
- Ingenieur (Nom, Prenom, CodeIngenieur, NomEntreprise, CodeEntreprise)
- Enseignant (Nom, Prenom, statut, CodeEnseignant)
- Cours (Intitule, CodeCours)
- Affectation (CodeEnseignant, CodeCours)
- Inscription (NSS, CodeCours)
- Tutorat (NSS, CodeEnseignant, CodeIngenieur)

En rouge sont marqués les attributs identifiants ou clés (spécifiés UNIQUE en SQL). Tous les attributs seront non NULL sauf Statut et Prenom.

Remarque : SQL ne distingue pas minuscule et majuscule (CodeIngenieur sera identique à codeingenieur). Les noms des tables et attributs ci-dessus commencent par une majuscule mais ceci n'indique pas qu'il font référence à une quelconque classe Java.

## Création de la base

```
import java.sql.*;
import java.io.*;
public class Creation{
  public static DataSource getSource() throws Exception {
     Context ctx = new InitialContext();
     DataSource source = (DataSource) ctx.lookup("jdbc/beal_base");
     return source;
  }
  public Creation() throws Exception{
     DataSource source = Creation.getSource();
     Connection co = source.getConnection();
     // lecture du fichier pour creer la base
     Statement st = co.createStatement();
     FileReader fichier = new FileReader("ir.sql");
     BufferedReader in = new BufferedReader(fichier);
     StringBuilder sb = new StringBuilder();
     String line;
     while ( (line = in.readLine()) != null) {
         if (line.equals("")) {
             st.executeUpdate(sb.toString());
             //System.out.println(sb.toString());
             sb = new StringBuilder();
         }
         else { sb.append(line+'\n'); }
     }
     in.close();
     st.close();
     co.close();
    }
   public static void main(String[] args) throws Exception{
       new Creation();
   }
}
```



Le fichier chargé (ir.sql) contient des requêtes SQL séparées par une ligne blanche.

```
CREATE TABLE Enseignant
   (Nom VARCHAR(25) NOT NULL,
    Prenom VARCHAR(25),
    Statut VARCHAR(25),
    CodeEnseignant INTEGER NOT NULL UNIQUE);

...

INSERT INTO Apprenti
VALUES('Dupont','Jacques',1450);

...
```

Vérification (ci-dessous par connexion directe à la base sous Unix) :

```
beal_base=> SELECT * FROM Enseignant;
   nom   |    prenom    |  statut   | codeenseignant
---------+--------------+-----------+----------------
 Beal    | Marie-Pierre | titulaire |           8001
 Roussel | Gilles       | titulaire |           8002
 Berstel | Jean         | titulaire |           8003
 Revuz   | Dominique    | titulaire |           8004
(4 rows)

beal_base=> SELECT * FROM Ingenieur;
    nom     | prenom  | codeingenieur |   nomentreprise   | codeentr
------------+---------+---------------+-------------------+----------
 Atome      | Michel  |          9202 | CEA               |    100
 Dechet     | Ludovic |          9203 | ANDRA             |    200
 Monge      | Gaspard |          9204 | CEA               |    100
 Banquetout | Felix   |          9205 | Credit-Avantageux |    300
(4 rows)

beal_base=> SELECT * FROM Apprenti;
  nom   |  prenom  | nss
--------+----------+------
```



```
 Dupont | Jacques  | 1450
 Dupond | Francois | 1451
 Tintin | David    | 1452
 Milou  | Jerome   | 1453
(4 rows)

beal_base=> SELECT * FROM Cours;
       intitule        | codecours
-----------------------+-----------
 Java                  | IR00
 Interfaces graphiques | IR01
 Systemes              | IR04
 Reseaux               | IR12
(4 rows)

beal_base=> SELECT * FROM Tutorat;
 nss  | codeenseignant | codeingenieur
------+----------------+---------------
 1450 |           8001 |          9202
 1451 |           8001 |          9204
 1452 |           8002 |          9205
 1453 |           8003 |          9202
(3 rows)

beal_base=> SELECT * FROM Affectation;
 codeenseignant | codecours
----------------+-----------
           8001 | IR00
           8003 | IR01
           8004 | IR04
           8004 | IR12
(4 rows)
```



## Exemple d'interrogation avec une requête statique

On demande les noms et prénoms de tous les apprentis dont le tuteur enseignant a pour nom `"Beal"`.

```
public class Interrogation{
  public Interrogation(Connection co) throws Exception{
     Statement st = co.createStatement();
     // donne les apprentis dont le tuteur enseignant est Beal
     String s ="SELECT Apprenti.Nom, Apprenti.Prenom "+
               "FROM Tutorat, Apprenti, Enseignant "+
               "WHERE Tutorat.CodeEnseignant  "+
               "= Enseignant.CodeEnseignant "+
               "AND Tutorat.NSS = Apprenti.NSS "+
               "AND Enseignant.Nom = 'Beal' ;";
     ResultSet rs = st.executeQuery(s);
     System.out.println("Mes tutes :");
     String n,p;
     while (rs.next()){
         n = rs.getString("Nom");
         p = rs.getString("Prenom");
         System.out.println(n+" "+p);
     }
     rs.close(); st.close();
    }
    public static void main(String[] args) throws Exception {
        DataSource source = Creation.getSource();
        Connection co = source.getConnection();
        Interrogation i = new Interrogation(co);
        if (co != null) co.close();
    }}
```

Le résultat est

```
Mes tutés :
Dupont Jacques
Dupond Francois
```

## Création d'une requête pré-compilée

La plupart des SGBDR permettent des requêtes pré-compilées. La méthode `PrepareStatement()` de la classe `Connection` renvoie un objet `PreparedStatement` où

- Les arguments qui seront passés de façon dynamique sont tous notés `?`.
- Ils sont ensuite positionnés par les méthodes `setInt(), setString(), ..., setNull()` de la classe `PreparedStatement` qui prennent en paramètres le numéro de l'argument (compté à partir de 1) et l'argument lui-même.
- Les requêtes sont ensuite lancées avec une des méthodes `execute` sans argument.

```
public PreparedInterrogation(String arg, Connection co)
throws Exception{
...
     String s =
        "SELECT Apprenti.Nom, Apprenti.Prenom "+
        "FROM Tutorat, Apprenti, Enseignant "+
        "WHERE Tutorat.CodeEnseignant = Enseignant.CodeEnseignant "+
        "AND Tutorat.NSS = Apprenti.NSS "+
        "AND Enseignant.Nom = ? ;";
     PreparedStatement ps = co.prepareStatement(s);
     ps.setString(1,arg);
     ResultSet rs = ps.executeQuery();
...
}
public static void main(String[] args) throws Exception{
  ...
  new PreparedInterrogation("Beal",co);
}
```



## Conservation de l'intégrité de la base

Afin de conserver l'intégrité de la base, certaines requêtes sont regroupées par paquets (appelés transactions) et ne sont validées que si toutes se sont exécutées normalement. Par défaut une requête constitue une transaction.

### *Validation des transactions*

- Par défaut une transaction est validée si l'exécution des requêtes (par `executeUpdate`) s'est bien passée. Par exemple, si une exception `SQLException` est levée, la transaction n'est pas validée.
- On peut changer ce mode par défaut par

  ```
  co.setAutoCommit(false);
  ```
- Pour valider les changements il faut alors le signaler explicitement par

  ```
  co.commit();
  ```
- On peut annuler les commandes non encore validées par un "commit" par :

  ```
  co.rollBack();
  ```

  L'état de la base juste après le dernier "commit" est restauré.



### *Exemple*

L'insertion d'un nouvel apprenti peut se faire par :

```
public Transaction(String nom, String prenom, int nss,
  int codeTE, int codeTI, Connection co) throws Exception{
     co.setAutoCommit(false);
     String s = "INSERT INTO Apprenti "+
             "VALUES(?,?,?) ;";
     PreparedStatement ps = co.prepareStatement(s);
     ps.setString(1,nom); ps.setString(2,prenom); ps.setInt(3,nss);
     ps.executeUpdate();
     s = "INSERT INTO Inscription "+
         "VALUES(?,'IR00') ;";
     ps = co.prepareStatement(s);
     ps.setInt(1,nss);
     ps.executeUpdate();
     s = "INSERT INTO Tutorat "+
         "VALUES(?, ?, ?) ;";
     ps = co.prepareStatement(s);
     ps.setInt(1,nss); ps.setInt(2,codeTE); ps.setInt(3,codeTI);
     ps.executeUpdate();
     // demande de transaction
     co.commit();
     co.setAutoCommit(true);
     ps.close();
   }
    // arrivée du nouvel apprenti Edgar Jacobs
    public static void main(String[] args) throws Exception {
        DataSource source = Creation.getSource();
        Connection co = source.getConnection();
        new Transaction("Jacobs","Edgar",1444, 8002, 9204);
        co.close();
    }
beal_base=# SELECT * FROM apprenti WHERE apprenti.nom = 'Jacobs';
  nom   | prenom | nss
--------+--------+------
 Jacobs | Edgar  | 1444
(1 row)
```



## Les "méta-données"

### *Méta-données sur les résultats*

- La méthode `ResultSetMetaData getMetaData()` de la classe `ResultSet` permet d'obtenir des informations sur les données renvoyées (et non ces données elles-mêmes).
- On peut ensuite obtenir
  - le nombre de colonnes : `getColumnCount()`,
  - le nom d'une colonne : `getColumnName(int column)`,
  - si NULL SQL peut être stocké dans une colonne : `int isNullable(int column)`.

### *Méta-données sur la base*

- Pour récupérer des informations sur la base elle-même dans son ensemble, on utilise la méthode `getMetaData()` de la classe `Connection`.
- Elle renvoie un objet de la classe `DatabaseMetaData`.
- Cette dernière interface possède des méthodes pour obtenir des informations sur la base.

***Exemple***

```
public class MetaInterrogation{
   public MetaInterrogation(Connection co) throws Exception{
     Statement st = co.createStatement();
     String s = "SELECT * from Ingenieur ;";
     ResultSet rs = st.executeQuery(s);
     ResultSetMetaData rsmd = rs.getMetaData();
     int nbColonnes = rsmd.getColumnCount();
     for (int i = 1; i <= nbColonnes; i++) {
         System.out.print(rsmd.getColumnName(i)+" ");
     }
     System.out.println();
     for (int i = 1; i <= nbColonnes; i++) {
         System.out.print(rsmd.isNullable(i)+" ");
     } System.out.println();
     System.out.print(ResultSetMetaData.columnNoNulls+" ");
     System.out.print(ResultSetMetaData.columnNullable+" ");
     System.out.println(ResultSetMetaData.columnNullableUnknown+" ");

     DatabaseMetaData dbmd = co.getMetaData();
     System.out.println(dbmd.getDatabaseProductName());
     System.out.println(dbmd.getURL());
     System.out.println(dbmd.getUserName());
     System.out.println(dbmd.getDriverName());
     System.out.println(dbmd.supportsPositionedUpdate());
     rs.close();  st.close();
    }}
```

```
$ java MetaInterrogation
nom prenom codeingenieur nomentreprise codeentreprise
0 1 0 0 0
0 1 2
PostgreSQL
jdbc:postgresql://localhost:5432/beal_base?prepareThreshold=0
beal
PostgreSQL Native Driver
false   // Cette base n'autorise pas les UPDATE positionnés.
```



## Modifications de la base à partir de résultats JDBC 2

Il est possible (JDBC 2, vieux) de modifier la base à partir des objets `ResultSet`.

```
public class ResultSetUpdate{
  public ResultSetUpdate(Connection co) throws Exception{
     String s = "SELECT * FROM Apprenti ; ";
     // résultats à modifier
     Statement st = co.createStatement(
                 ResultSet.TYPE_SCROLL_INSENSITIVE,
                 ResultSet.CONCUR_UPDATABLE);
     ResultSet rs = st.executeQuery(s);
     rs.absolute(5);
     System.out.println(rs.getString(2)); // Edgar
     rs.updateString(2, "Edgar P."); //change la colonne 2
     rs.updateRow();      // mise à jour dans la base
     rs.close();
     st.close();
    }

    public static void main(String[] args) throws Exception {
       DataSource source = Creation.getSource();
       Connection co = null;
       try {
         co = source.getConnection();
         new ResultSetUpdate(co);
       } catch (Exception e) {
       e.printStackTrace();
       }
       if (co != null) co.close();
    }
}
```



## Modifications de la base à partir de résultats JDBC 3

- Un `RowSet` est un objet contenant un ensemble de lignes provenant d'un `ResultSet`.
- Il existe deux types de `RowSet`, ceux qui sont en permanence connectés à la base, comme `JDBCRowSet`, et ceux qui ne se connectent à la base que lorsque c'est nécessaire, comme `CachedRowSet`.
- Ainsi un `CachedRowSet` effectue les modifications dans un tampon avant de les commuter sur la base via un driver ou une `DataSource`. Il garde aussi une copie de la zone avant les modifications pour tester s'il n'y a pas eu de modifications concurrentes pendant qu'il était déconnecté.

```
import java.sql.*;
import javax.sql.*;
import javax.sql.rowset.*;
import java.io.*;
import org.postgresql.ds.*;
import com.sun.rowset.*;
```



```
public class CachedRowSetUpdate {

  public CachedRowSetUpdate throws Exception {
     CachedRowSetImpl crs = null;
     try {
     crs = new CachedRowSetImpl(); // dans com.sun.rowset
     crs.setType(ResultSet.TYPE_SCROLL_INSENSITIVE);
     crs.setConcurrency(ResultSet.CONCUR_UPDATABLE);
     String s = "SELECT * FROM Apprenti ; ";
     crs.setCommand(s);
     crs.setDataSourceName(java/beal_base");  // géré par JNDI;
     crs.setUsername("beal");
     crs.setPassword("");
     crs.absolute(5);
     crs.updateString(2, "Edgar P.");
     crs.updateRow(); // mise à jour dans le RowSet
     crs.acceptChanges(); // mise à jour dans la base

     } catch (SQLException e) {
     System.out.println(e.getMessage());
     }  catch (Exception e) {
     e.printStackTrace();
     } finally {
     if (crs != null) crs.close();
     }
  }


    public static void main(String[] args) throws Exception {
       new CachedRowSetUpdate();
    }
}
```

## Introspection

La classe `java.lang.Class<T>` :

- permet de manipuler les classes et interfaces comme des objets;
- offre des possibilités d'introspection (exploration des méthodes et constructeurs d'une classe).

On peut ensuite récupérer un objet "méthode". Les classes de ces objets sont définies dans le paquetage `java.lang.reflect` :

```
class Test {
    public static void main(String[] args) {
        // La classe Class<T> représente la classe d'un objet
        // de type T.
        Class<Float> c1 = float.class; System.out.println(c1);
        Class<Float> c2 = Float.class; System.out.println(c2);
        Class<List> c3 = java.util.List.class; System.out.println(c3
        // java.util.List<Integer>.class; erreur
        Class<Cloneable> c4 = Cloneable.class; System.out.println(c4

        // o.getClass() renvoie un objet de type Class<? extends E>
        // où E est le type abrasion du type de o.
        String s = "toto";
        Class<? extends String> c5 = s.getClass();
        System.out.println(c5);
        List<Integer> l = new ArrayList<Integer>();
        Class<? extends List> c6 = l.getClass();
        try {
        Class<?> c7 = Class.forName("java.lang.Cloneable");
        System.out.println(c7);
        } catch (ClassNotFoundException e) {
        }
    }
}
```



```
float
class java.lang.Float
interface java.util.List
interface java.lang.Cloneable
class java.lang.String
interface java.lang.Cloneable
```

Une instance de la classe `Class<T>` est associée à toutes les classes, interfaces, tableaux ou types primitifs.

On peut appliquer les méthodes suivantes à un objet `c` de la classe `Class<T>`

- `getDeclaredMethods()` retourne un tableau d'objets de la classe `java.lang.reflect.Method`, les méthodes déclarées dans `c`;
- `getMethods()` retourne aussi les méthodes héritées;
- `getMethods(String, Class... parameterTypes)` recherche une méthode en fonction de son profil.

Une méthode de la classe `Method` peut ensuite être invoquée par `invoke()`.



```
import java.lang.reflect.*;
import java.util.Date;

public class MyDate{
  public static void main(String[] args) throws Exception {
    Class<?> classeDate = Class.forName("java.util.Date"); // ou "Da
    Object myDate = classeDate.newInstance();
      //une instance de la classe Date est créée
    Method myOutput =
       classeDate.getMethod("toString", (Class[])null);
      //retourne la methode toString() de la classe Date
    System.out.println(
       (String) myOutput.invoke(myDate, (Object[])null));
      //appel de toString() sur myDate
    Date today = new Date();
    System.out.println(today);
    System.out.println(myDate);
  }
}
```

On obtient :

```
> java Test
Wed Mar 30 10:09:32 GMT 2005
Wed Mar 30 10:09:32 GMT 2005
Wed Mar 30 10:09:32 GMT 2005

import java.lang.reflect.*;
import java.util.Date;

public static void main(String[] args) throws Exception{

  Class<Date> classeDate = Date.class;
  Date myDate = classeDate.newInstance();
     // une instance de la classe Date est crée
  Method myOutput =
     classeDate.getMethod("toString", (Class[])null);
     // on  recupere la methode toString() de la classe Date
  System.out.println(
```



```
     (String) myOutput.invoke(myDate,(Object[])null));
     // appel de toString() sur myDate
  Date today = new Date();
  System.out.println(today);
  System.out.println(myDate);
}
```

Soit la classe `MyClasse`

```
public class MyClasse {
  public static void toto(int i){
     System.out.println(i);
 }
}
```

Et la classe `Test`

```
import java.lang.reflect.Method;
public class Test {
        public static void main(String[] args) {
               Class<MyClasse> c = MyClasse.class;
               System.out.println(c);
               Method[] t=c.getDeclaredMethods();
               for(int i=0; i< t.length; i++){
                       System.out.println(t[i]);
               }
               try{
                   Method m = c.getMethod("toto", int.class);
                   Integer i = 100;
                   MyClasse o = c.newInstance();
                   m.invoke(o,i);
                   MyClasse.toto(i);
                   System.out.println(i);
               }
               catch(Exception e){//Nothing
                   System.out.println(e);
               }
        }
}
```

```
class MyClasse
public static void MyClasse.toto(int)
100
100
100
```



## Le chargement de classes

Un *chargeur de classes* (*classloader*) est une instance d'une sous-classe de la classe abstraite `java.lang.ClassLoader`. Il charge le bytecode d'une classe à partir d'un fichier `.class` et la rend accessible aux autres classes.

Principe du fonctionnement de la méthode `loadClass()` de la classe `ClassLoader` :

1. appel à `findLoadedClass()` pour voir si la classe n'est pas dejà chargée;
2. demande de chargement de la classe à un chargeur parent obtenu par `getParent()`;
3. en cas d'échec, appel de la méthode `findClass()`;
4. levée de l'exception `ClassNotFoundException` en cas de nouvel échec.



```
public Class<?> loadClass(String name)
       throws ClassNotFoundException {
  try {
    Class<?> c = findLoadedClass(name);
    if (c != null) return c;

    ClassLoader parent = getParent();
    try {
      c = parent.loadClass(name);
      if (c != null) return c;
    } catch (ClassNotFoundException e) {}

    c = findClass(name);
    if (c !=null) return c;
  } catch (Exception e) {
      throw new ClassNotFoundException(name);
    }
}
```

La méthode `findClass()` appelle une méthode `defineClass()` qui est la méthode de base de tout chargeur de classes. Elle

- crée une instance de la classe `Class`
- stocke la classe dans le chargeur

La signature de `defineClass()` est :

```
Class<?> defineClass(String name,byte[] b,int off,int len
                throws ClassFormatError
```



```
import java.io.*;

public class VerboseClassLoader extends ClassLoader{
  public VerboseClassLoader(){
    super(getSystemClassLoader());//chargeur parent en param.
  }
  public Class<?> loadClass(String name)
    throws ClassNotFoundException {
    System.out.println("Chargement de "+ name);
    try {
      byte[] b = loadClassData(new File(name+".class"));
      return defineClass(name,b,0,b.length);
    } catch (Exception e) {
        return getParent().loadClass(name);
      }
  }
  private byte[] loadClassData(File f) throws IOException {
    FileInputStream entree = new FileInputStream(f);
    int length = (int)f.length();
    int offset = 0;
    int nb;
    byte[] tab = new byte[length];
    while (length != 0) {
      nb = entree.read(tab,offset,length);
      length -= nb;
      offset += nb;
    }
    return tab;
  }
}
```

```
    public static void main(String[] args) throws Exception{
      VerboseClassLoader cl = new VerboseClassLoader();
      Class<?> clazz = cl.loadClass("A");
      Object o = clazz.newInstance();
      System.out.print("Dans VerboseClassLoader : ");
      if (o instanceof A)
        System.out.println("o instance de A");
      else
        System.out.println("o n'est pas instance de A");
      System.out.println((o.getClass()).getClassLoader());
      A o2 = new A();
      System.out.println((o2.getClass()).getClassLoader());
    }
  }
```

Étant données trois classes vides `B`,`C` et `D`, la classe `A` est :

```
public class A extends B {
  C c;
  D d;

  public A() {
    System.out.println("nouveau A()");
    d = new D();
  }

  public void inutile(){
    c = new C();
  }
}
```



Dans l'exemple précédent,

- l'appel à `newInstance()` crée un objet et charge les classes nécessaires à sa création;
- `o` et `o2` *n'appartiennent pas* à la même classe : deux classes de même nom (ici `A`) peuvent coexister dans la machine virtuelle si elles n'ont pas le même chargeur de classe. Ceci est important pour la *programmation réseau*.

On obtient :

```
> java VerboseClassLoader
Chargement de A
Chargement de B
Chargement de java.lang.Object
Chargement de java.lang.System
Chargement de java.io.PrintStream
nouveau A()
Chargement de D
Dans VerboseClassLoader : o n'est pas instance de A
VerboseClassLoader@7d772e
nouveau A()
sun.misc.Launcher$AppClassLoader@94af67
```



# 6

# La programmation concurrente





## Programmation concurrente

La *programmation concurrente* est l'ensemble des mécanismes permettant l'exécution concurrente d'actions spécifiées de façon séquentielle.
En Java, deux mécanismes permettent un ordonnancement automatique des traitements :

- la concurrence entre commandes du système (processus);
- la concurrence entre processus légers de la machine virtuelle.

## Processus légers

Un *processus léger* (*thread*) correspond à un fil d'exécution (une suite d'instruction en cours d'exécution). Il s'agit d'un processus créé et géré par la machine virtuelle Java.

S'il y a plusieurs processus légers :

- ils sont associés à un même programme;
- ils s'exécutent dans le même espace mémoire.

Lorsque l'on parle de processus léger, il y a trois notions bien distinctes :

- un objet représentant le code à exécuter (la cible).
  La classe de cet objet implémente l'interface `Runnable`.
- un objet qui contrôle du processus léger.
  Il est d'une classe dérivant de `Thread`.
- un fil d'exécution, c'est-à-dire la séquence d'instructions en cours d'exécution.
  C'est le code de la méthode `run()` de la cible.

Ne pas confondre `Thread` (le contrôleur) et `Runnable` (le contrôlé). C'est d'autant plus facile que `Thread` implémente `Runnable` et peut donc s'autocontrôler !



## La classe `java.lang.Thread`

- Un objet de la classe `Thread` ne représente pas un processus léger mais un *objet de contrôle du processus léger*.
- Au lancement d'un programme, la machine virtuelle possède un unique processus léger qui exécute le `main()` de la classe appelée.

```
public class MyThread{
  public static void main(String[] args) throws Exception{
    Thread threadInitiale = Thread.currentThread();
    threadInitiale.setName("Thread initiale");
    System.out.println(threadInitiale);
    Thread.sleep(1000);
    System.out.println(threadInitiale.isAlive());
    Thread myThread = new Thread();
    maThread.setName("Ma thread");
    System.out.println(myThread);
    System.out.println(myThread.isAlive());
  }
}
```

On obtient à l'exécution :

```
Thread[Thread initiale,5,main]
true
Thread[Ma thread,5,main]
false
```

Chaque processus léger appartient à un groupe de processus légers (ici `main`) et a une priorité (ici 5).



## Démarrage et terminaison

- **Démarrage** d'un processus léger par la méthode `start()` du thread.
- **Exécution** du processus léger par le thread qui appelle la méthode `run()` du runnable.

La méthode `run()` peut être spécifiée de deux façons différentes :

- en implémentant la méthode `run()` de l'interface `Runnable` (solution explicite).
- en redéfinissant la méthode `run()` de la classe `Thread` (solution directe).

Le processus léger se termine à la fin du `run()`.

La classe `Thread` possède sept constructeurs qui spécifient :

- le nom du processus léger (par défaut `Thread-i`);
- le groupe du processus léger (un objet de la classe `ThreadGroup`);
- la cible (target) du processus léger : un objet implémentant l'interface `Runnable` qui précise la méthode `run()` à exécuter lors du démarrage du processus léger.



## Le lapin et la tortue (première version)

Classe des lapins

```
public class Lapin implements Runnable{
  public void run() {
    long t = System.currentTimeMillis(), x = t;
    for (int i = 0; i<5; i++){
      x = System.currentTimeMillis();
      System.out.println("Lapin "+i
        + " au temps "+ (x-t) + " ms.");
      try {
        Thread.sleep(300);  // il se repose peu
      } catch(InterruptedException e) {}
    }
    x = System.currentTimeMillis();
    System.out.println("Lapin est arrivé au temps "
      + (x-t) + " ms.");
  }
}
```

Classe des tortues

```
public class Tortue implements Runnable{
  public void run() {
    long t = System.currentTimeMillis(), x = t;
    for (int i = 0; i<5 ; i++){
      x = System.currentTimeMillis();
      System.out.println("Tortue "+i
        + " au temps "+ (x-t) + " ms.");
      try {
        Thread.sleep(500);   // il se repose beaucoup
      } catch(InterruptedException e) {}
    }
    x = System.currentTimeMillis();
    System.out.println("Tortue est arrivée au temps "
      + (x-t) + " ms.");
  }
}
```

Mise en place

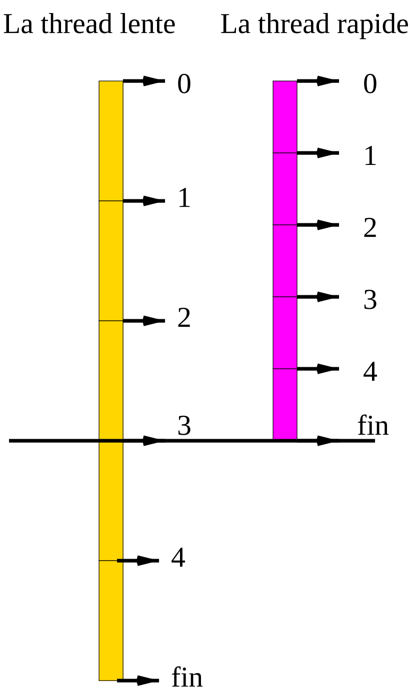
```
public class MesThread{
  public static void main(String[] args){
    Runnable tortue = new Tortue();
    Runnable lapin = new Lapin();

    Thread tortueThread = new Thread(tortue);
    Thread lapinThread = new Thread(lapin);

    tortueThread.start();
    lapinThread.start();
  }
}
```

On obtient :

```
Tortue 0 au temps 0 ms.
Lapin 0 au temps 0 ms.
Lapin 1 au temps 302 ms.
Tortue 1 au temps 515 ms.
Lapin 2 au temps 612 ms.
Lapin 3 au temps 922 ms.
Tortue 2 au temps 1025 ms.
Lapin 4 au temps 1232 ms.
Tortue 3 au temps 1535 ms.
Lapin est arrivé au temps 1542 ms.
Tortue 4 au temps 2045 ms.
Tortue est arrivée au temps 2555 ms.
```

Attention : Ici, les deux thread ont même priorité, donc même accès au processeur (équité). L'équité d'accès au processeur n'est pas assurée sur toutes les implémentations des machines virtuelles.








### Le lapin et la tortue (deuxième version)

Classe des lapins étend Thread :

```
public class Lapin extends Thread {
  public void run() {
    // inchangé
  }
}
```

Classe des tortues étend Thread :

```
public class Tortue extends Thread {
  public void run() {
    // inchangé
  }
}
```

Mise en place

```
public class MesThread{
  public static void main(String[] args){
    Thread tortueThread = new Tortue();
    Thread lapinThread = new Lapin();

    tortueThread.start();
    lapinThread.start();
  }
}
```



### Terminaison d'un processus léger

Terminaison normale : à la fin de la méthode `run()`.

On peut forcer la terminaison d'un processus léger avant la fin du `run()` en terminant l'application. L'application se termine lorsque :

- `Runtime.exit()` est appelée par l'un des processus légers;
- tous les processus légers qui n'ont pas été marqués *daemon* sont terminés.

Un processus léger peut être *user* ou *daemon*. On peut créer des processus légers *daemon* grâce à la méthode `setDaemon()` de la classe `Thread`. (Ex. `maThread.setDaemon(true);`).

*Priorités d'accès au processeur*

Les niveaux de priorité d'accès au processeur varient de 1 à 10. Des constantes de la classe `Thread` les définissent :

| | |
|---|---|
| `Thread.MAX_PRIORITY` | 10 |
| `Thread.NORM_PRIORITY` | 5 |
| `Thread.MIN_PRIORITY` | 1 |

On peut définir et consulter un niveau de priorité en appliquant une des méthodes suivantes sur l'objet de contrôle du processus léger :

- `setPriority()`
- `getPriority()`
- `setMaxPriority()`

## Exclusion mutuelle

Une *opération atomique* est une opération qui ne peut être interrompue une fois qu'elle a commencé.

- Java garantit l'atomicité de l'accès et de l'affectation des variables de type primitif (*sauf* `long` *et* `double`).
- Java possède un mécanisme d'*exclusion mutuelle* entre processus légers. Il garantit l'atomicité d'exécution de morceaux de code.

Un *verrou* peut être associé à une portion de code et permet d'exclure l'accès de deux processus légers sur cette portion.

Pour cela, on *synchronise* une portion de code relativement à un objet en utilisant le mot clé `synchronized` :

- `synchronized` comme modificateur d'une méthode : s'applique au code d'une méthode relativement à l'objet courant.
- `synchronized(obj){... portion de code ...}`;

Pendant l'éxécution par un processus léger $A$ d'une portion de code `synchronized`, tout autre processus léger essayant d'exécuter une portion de code `synchronized` relative au même objet est suspendu. Une fois $A$ terminé, un seul des processus légers en attente est relancé.



## Exemple : tableau

```
public class Table{
  private int[] tab;

  public synchronized int somme(){
    int s = 0;
    for (int i = 0 ; i < tab.length ; i++)
      s += tab[i];
    return s;
  }

  public synchronized void setElem(int i, int j){
    tab[i] = j;
  }
}
```

Pendant l'exécution d'un `x.setElem()` ou d'un `x.somme()` dans un processus `P`, tout autre processus `Q` qui essaie de faire `x.setElem()` ou `x.somme()` sur le même `x` est suspendu.



## Exemple : variante

```
public class Table{
  private int[] tab;

  public synchronized int somme(){
    int s = 0;
    for (int i = 0; i < tab.length ; i++)
      s += tab[i];
    return s;
  }

  public void setElem(int i, int j){
    if (i < 0 || i>= tab.length)
      throw new IndexOutOfBoundsException();
    synchronized(this) {
      tab[i] = j;
    }
  }
}
```

Dans cette version, seule l'affectation est vérouillée : un processus qui essaie d'écrire à un index hors bornes n'est pas suspendu.



## Sûreté et vivacité

Quelques notions :

- *sûreté (safety)* rien de faux peut se produire. L'exclusion mutuelle règle le problème de l'accès concurrent en écriture.
- *vivacité (liveness)* tout processus peut s'exécuter.

Les diverses facettes de la non vivacité :

- famine (contention): un processus léger est empêché de s'exécuter parce que un processus plus prioritaire accapare le processeur;
- endormissement (dormancy) : un processus léger est suspendu et jamais réveillé;
- terminaison prématurée;
- interblocage (deadlock) : plusieurs processus légers s'attendent mutuellement avant de continuer.

## Synchronisation entre processus légers

Java propose deux mécanismes :

- attente/notification avec `wait()` et `notify()`
  `wait()` appelé sur un objet suspend le processus léger courant qui attend une notification d'un autre processus via le moniteur de l'objet;
  `notify()` appelé sur un objet libère un processus léger en attente par `wait()` sur le moniteur du même objet.
- attente de terminaison avec `join()`
  `join()` est appelé sur l'objet de contrôle d'un processus léger dont la terminaison est attendue; le processus courant est alors interrompu jusqu'à la fin de celui-ci.

Les méthodes `wait()`, `join()` et `sleep()` peuvent être interrompues (et leur processus est alors débloqué). La méthode bloquante lève une exception `InterruptedException` qui peut être captée.



## Exemple : les tourneurs et le compteur

Cinq processus légers (les `Turner`) veulent faire tourner un compteur (le `Counter`) qui compte modulo 5.
Voici le compteur:

```
public class Counter {
  private int max; // 5 dans l'exemple
  private int count = 0; // initialisation, importante !
  public Counter (int max) {
    this.max = max;
  }
  public int getMax(){
    return max;
  }
  public int getValue(){
    return count;
  }
  public synchronized void increment(){
    count = (count +1) % max ; // ce qu'un tourneur veut faire
  }
}
```

La règle du jeu : un Tourneur ne peut faire tourner le compteur que lorsqu'il est égal à son numéro.



Voici la classe des tourneurs.

```
public class Turner extends Thread{
  private Counter c;  // le compteur
  private int numero;  // numéro du tourneur
  public Turner(int numero, Counter c){
    System.out.println("Tourneur "+ numero + " créé.");
    this.numero = numero;
    this.c = c;
    if (numero + 1 < c.getMax()) {
      new Turner(numero + 1, c);
    }
    System.out.println("Tourneur "+ numero + " démarre.");
    start();
  }

  public void run(){}

  public static void main(String[] args) {
    Counter c = new Counter(5);
    new Turner(0, c);
  }
}
```

Début d'exécution

```
Tourneur 0 créé.
Tourneur 1 créé.
Tourneur 2 créé.
Tourneur 3 créé.
Tourneur 4 créé.
Tourneur 4 démarre.
Tourneur 3 démarre.
Tourneur 2 démarre.
Tourneur 1 démarre.
Tourneur 0 démarre.
```



```
  public void run()
    try {
      for (int etape = 0 ; ; etape++) {
        System.out.println("Tourneur "+ numero
          + " in étape "+ etape);
        synchronized(c){
          while (numero != c.getValue())
            c.wait(); // suspend this
          System.out.println("Tourneur "+ numero
            + " out étape "+ etape);
          c.increment();
          c.notifyAll();  // libère les threads supendus
        }
      }
    } catch (InterruptedException e) {}
  }
```

Et le résultat:

```
...
Tourneur 4 démarre.
Tourneur 4 in étape 0
Tourneur 3 démarre.
Tourneur 3 in étape 0
Tourneur 2 démarre.
Tourneur 2 in étape 0
Tourneur 1 démarre.
Tourneur 1 in étape 0
Tourneur 0 démarre.
Tourneur 0 in étape 0
Tourneur 0 out étape 0
Tourneur 1 out étape 0
Tourneur 2 out étape 0
Tourneur 3 out étape 0
Tourneur 4 out étape 0
Tourneur 4 in étape 1
Tourneur 3 in étape 1
...
```

## Maître et esclave

Un esclave "travaille"

```
public class Slave implements Runnable {
  private int result;
  public int getResult(){ return result; }
  public int hardWork(){ return 0; }
  public void run(){ result = hardWork(); }
}
```

Le maître fait travailler l'esclave, et attend, par `join`, la fin du processus esclave.

```
public class Master implements Runnable{
  public void run(){
    Slave e = new Slave();
    Thread slave = new Thread(e);
    slave.start();
    // fait autre chose
    try {
      slave.join(); // attente de fin du run
    } catch (InterruptedException ex){}
    int result = e.getResult();
    System.out.println(result);
  }
}
```

Mise en place:

```
public class MasterTest{
  public static void main(String[] args) {
    Master m = new Master();
    Thread master = new Thread(m);
    master.start();
  }
}
```



## Variables locales à un processus léger

On peut simuler des variables locales à chaque processus léger. Pour cela :

- on crée un objet de la classe `ThreadLocal`;
- on y accède par `Object get()`;
- on le modifie par `void set(Object o)`.

*Exemple*

```
public class MyTarget implements Runnable {
    public ThreadLocal v = new ThreadLocal();
    public void run() {
      v.set(new Double(Math.random()));
      System.out.println(v.get());
    }
    public static void main(String[] args){
      MyTarget c = new MyTarget();
      Thread t1 = new Thread(c);
      Thread t2 = new Thread(c);
      t1.start();
      t2.start();
    }
}
```

On obtient :

```
0.8955847189505597
0.43636788900311063
```