Programmation objet et JAVA

0. Bibliographie
1. Programmation objet
2. Le langage Java 1.8
3. Quelques classes fondamentales (API)
4. Les flots

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. Premiers exemples

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 1.8 (aussi appelée Java 8). 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 des points

```
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);
  }
}
```

2

Le langage Java 1.8

1. Structure d'un programme
2. Classes et objets
3. Méthodes et constructeurs
4. Visibilité et paquetage
5. Types primitifs et enveloppes
6. Sous-typage

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`.

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
    }
  }
}
```

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 !

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 :

```
public class Pixel {
  private 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;
}
```

Exemple avec un constructeur particulier :

```
public class Pixel {
  private int x,y;
  public 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.

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

```
public class Segment {
  private Pixel start ;
  private 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 :

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

```
    return start.toString()+"--"+end.toString();
  }
}
```

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) {
  Pixel p1 = new Pixel(2,3);
  Pixel p2 = new Pixel(5,8);
  Segment s = new Segment(p1,p2);
  Segment t = new Segment(2,3,5,8);
  System.out.println(s); //(2,3)--(5,8)
  System.out.println(t); //(2,3)--(5,8)
}

public static void main(String[] args) {
  Pixel p1 = new Pixel(2,3);
  Pixel p2 = null;
  Segment s = new Segment(p1,p2);
  System.out.println(s.start); //(2,3)
  System.out.println(s.end);  // NullPointerException
}
```

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`.

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.

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.

```
public class Pixel {
   private int x, y;
   public 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());
  }
}
```

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.

3

Délégation

1. Exemple de délégation

Points et disques : exemple de délégation

Il ne faut pas confondre dérivation et délégation.

On considère une classe `Disk` pour représenter des disques. Chaque disque contient un centre de type `Pixel` et un rayon

```
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
  }
}
```

4

Le langage Java 1.8 suite

1. Les tableaux
2. Exemple avec une documentation

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()
*/
```

5

Héritage et exceptions

1. Héritage : généralités
2. Héritage : exemples
3. Héritage : interfaces
4. Exceptions
5. Exemples de types paramétrés

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 banquaire.
- 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.

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*.

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
```

Le transtypage

Le transtypage donne la possibilité de référencer un objet d'un certain type avec un autre type.

- 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 parle de sous-typage en Java quand une variable de type B contient implicitement une référence à un objet d'une classe dérivée D de B. Ce sous-typage est implicite.

- 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 `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
```

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.

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
if (null instanceof Object) {...} // false
```

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

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];
  }
}

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();
    }
  }
}
```

On obtient

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

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.

6

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
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`.

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 implements Shape{
  private final int width;
  private 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).hashCode();
  }
```

ou bien

```
  @Override
  public int hashCode() {
    return Objects.hash(width,height);
  }
```

On utilise ici la méthode de la classe outils `java.util.Objects`

```
public static int hash(Object... values)
```

Attention, appelée avec un seul argument, elle ne renvoie pas le hash code de l'argument.

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.

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.

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);
```

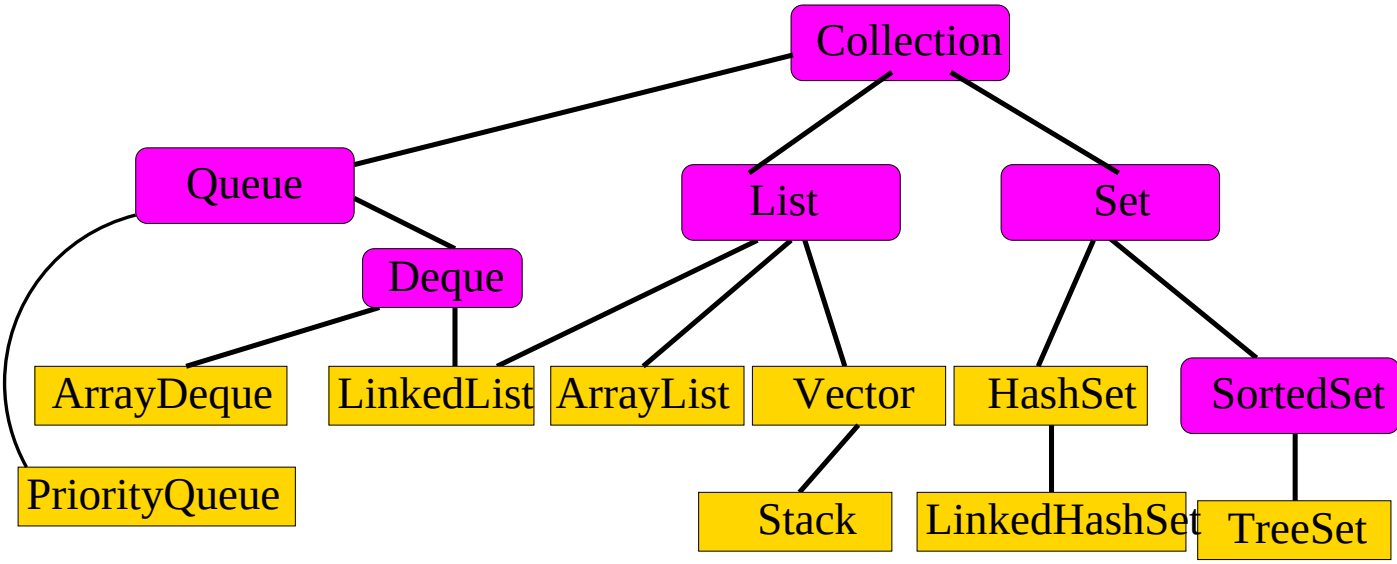
- Une instruction arithmétique sur les entiers peut lever l'exception `ArithmeticException` :

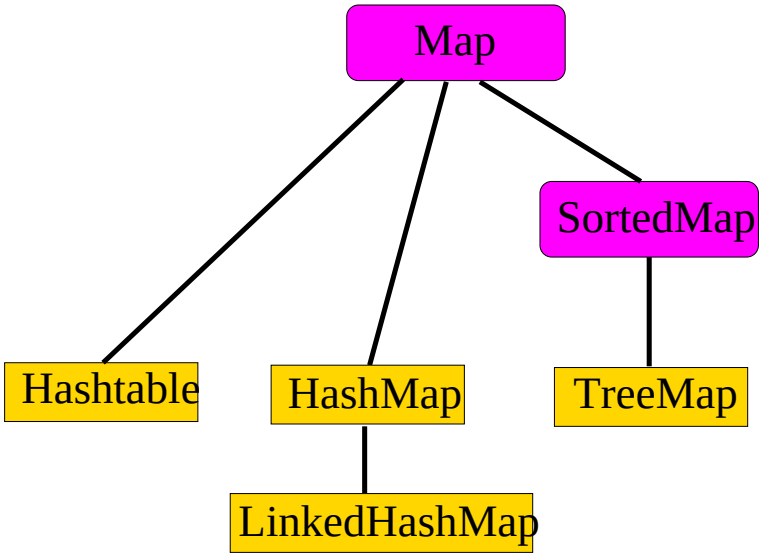
```
try { int i = 1/0;}
catch (ArithmeticException e) {...}
```

- 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





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, trois classes d'utilitaires

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

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>`.

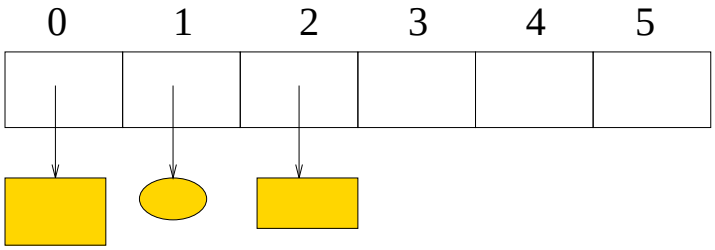
Exemple de ArrayList

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* .

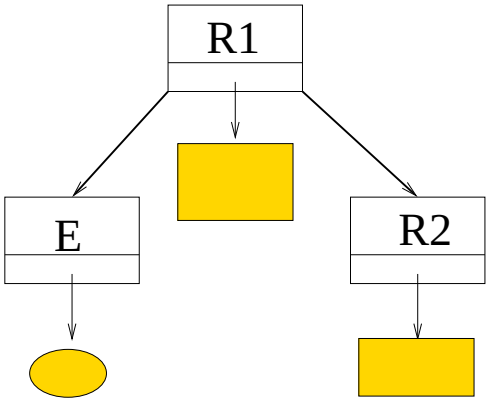
L'interface `Shape`, et les classes `Rectangle` et `Ellipse` sont comme d'habitude.

```
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
```

7

Les flots

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

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.URI`).
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

```
Reader
   BufferedReader
      LineNumberReader
   CharArrayReader
   FilterReader
      PushbackReader
   InputStreamReader
      FileReader
   PipedReader
   StringReader
```

Writer

```
Writer
   BufferedWriter
   CharArrayWriter

   FilterWriter

   OutputStreamWriter
      FileWriter
   PipedWriter
   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 bufférisér 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.toURI());
    } 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
```