Programmation objet et JAVA

0

Bibliographie

Livre de référence

- Gilles Roussel, Étienne Duris, *Java et Internet : concepts et programmation*, Vuibert, 2000.

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

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.

Un exemple

```
class Point {
  private int x;
  private int y;

  public Point (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 déplace(int dx, int dy) {
    x += dx; y += dy;
  }
  public String toString() {
    return(this.x + ", " + this.y);
  }

  public static void main(String[] args) {
    Point a = new Point(3,5);
    a.setY(6);      // a = (3,6)
    a.déplace(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

```
class Disque {
  protected Point centre;  // composition
  protected int rayon;
  public Disque(int x, int y, int rayon) {
    centre = new Point(x,y);
    this.rayon = rayon;
  }
  public String toString() {
    return centre.toString() + " ," + rayon;
  }
  public void déplace(int dx, int dy) {
    centre.déplace(dx, dy);  // délégation
  }
}

class Anneau extends Disque { // dérivation
  private int rayonInterne;
  public Anneau(int x, int y, int rayon, int rayonInterne) {
    super(x, y, rayon);
    this.rayonInterne = rayonInterne;
  }
  public String toString() {
    return super.toString() + " ," + rayonInterne;
  }
}

class TestDisqueAnneau {
  public static void main(String[] args) {
    Anneau a = new Anneau(3, 5, 7, 2);
    a.déplace(5,-2); // héritée de Disque, qui délègue à Point
    System.out.println(a); // 8, 3 , 7, 2
  }
}
```


2

Le langage Java

Premier exemple

Le fichier `HelloWorld.java` :

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

Compilation : `javac HelloWorld.java`

crée le fichier `HelloWorld.class`

Exécution : `java HelloWorld`

Résultat : `Coucou !`

- 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`.

Deuxième exemple

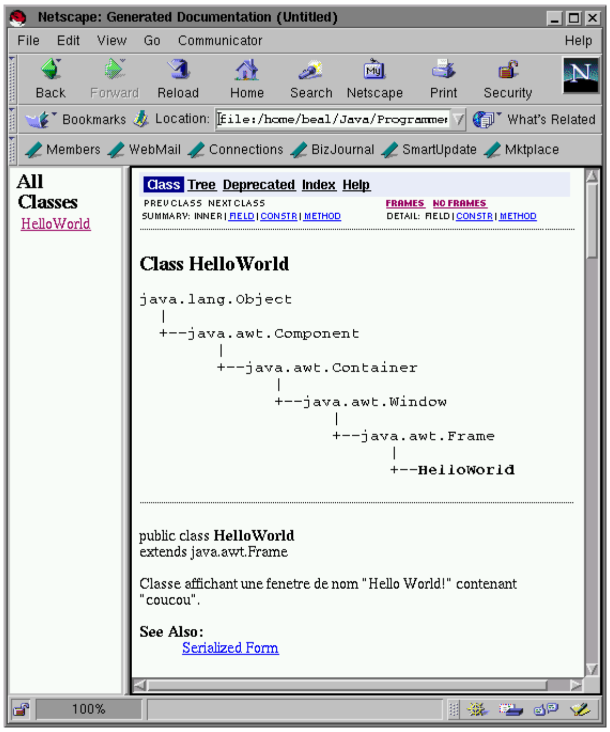
```
import java.awt.Frame;
import java.awt.Graphics;
import java.awt.Color;
/**
 * Classe affichant une fenetre de nom "Hello World"
 * contenant "coucou".
 * @author Beal
 * @version 1.0
 */
public class HelloWorld extends Frame{
  /**
   * Constructeur
   */
  public HelloWorld() {
    super("Hello World");
    setSize(200,100);
    show();
  }
  /**
   * Methode de dessin de la fenetre
   * @param graphics contexte d'affichage
   */
  public void paint(Graphics graphics) {
    graphics.setColor(Color.black);
    graphics.drawString("coucou",65,60);
  }
  /**
   * Methode principale
   * @param args arguments de la ligne de commande
   */
  public static void main(String[] args) {
    new HelloWorld();
  }
}
```

Création de la documentation HTML

```
javadoc HelloWorld
```

Visualisation avec Netscape.

```
$ ls

HelloWorld.class        index.html
HelloWorld.html         overview-tree.html
HelloWorld.java         package-list
allclasses-frame.html   packages.html
deprecated-list.html    serialized-form.html
help-doc.html           stylesheet.css
index-all.html
```

Créer une documentation HTML

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

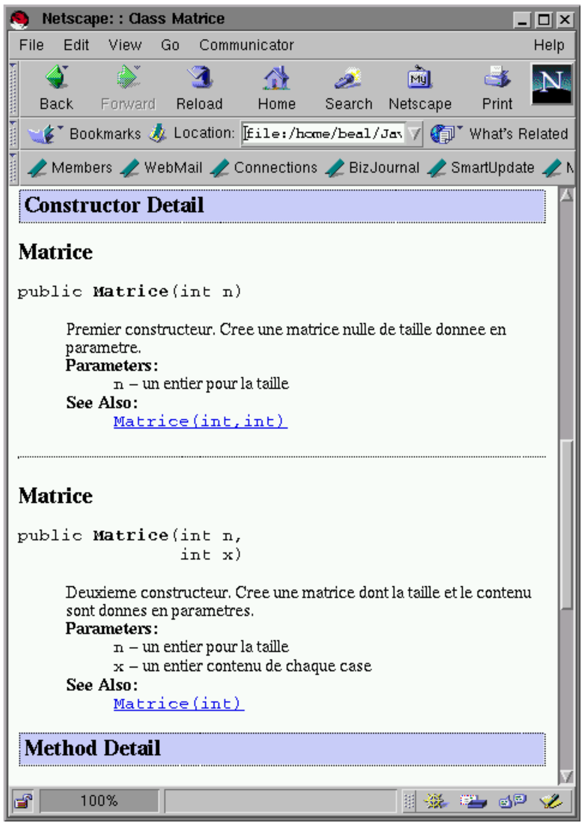
- `@author` : il peut y avoir plusieurs auteurs;
- `@see` : pour créer un lien sur une autre documentation Java;
- `@param` : pour indiquer les paramètres d'une méthode;
- `@return` : pour indiquer la valeur de retour d'une méthode;
- `@exception` : pour indiquer quelle exception est levée;
- `@version` : 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()
*/
```


Un exemple :

```
/**
 * Classe permettant de manipuler des matrices.
 * @author Beal
 * @author Berstel
 * @version 1.0
 */

public class Matrice {
  int n;
  int[][] m;

  /**
   * Premier constructeur. Cree une matrice
   * nulle de taille donnee en parametre.
   * @param n un entier pour la taille
   * @see Matrice#Matrice(int,int)
   */

  public Matrice(int n) {
    this.n = n;
    m = new int [n][n];
  }

  /**
   * Deuxieme constructeur. Cree une matrice
   * dont la taille et le contenu sont donnes
   * en parametres.
   * @param n un entier pour la taille
   * @param x un entier contenu de chaque case
   * @see Matrice#Matrice(int)
   */

  public Matrice(int n, int x) {
    this.n = n;
    m = new int [n][n];
    for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        {m[i][j]=x;}
  }
```

```
/**
 * Methode de transposition d'une matrice
 */

public void transposer() {
  for (int i = 0; i < n; i++)
  for (int j = i+1; j < n; j++)
  { int t = m[i][j]; m[i][j] = m[j][i]; m[j][i]=t;}
}

/**
 * Methode d'affichage d'une matrice
 */

public String toString() {
  StringBuffer sb = new StringBuffer();
  for (int i = 0; i < n; i++)
  { for (int j = 0; j < n; j++)
    {sb.append(m[i][j]);
    sb.append(" ");}
  sb.append("\n");}
  return sb.toString();
}

/**
 * Methode principale
 * @param args arguments de la ligne de commande
 */

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

Fractales

Principe : objets définis récursivement.
Exemple : la courbe du dragon de Heighway, qui est donnée par l'applette suivante :

```
import java.awt.*;
import java.applet.*;

public class Dragon extends Applet {
  public void paint(Graphics g) {
    g.setColor(Color.red);
    drawDragon(g, 20, 100, 100, 200, 200);
  }
  private void drawDragon (Graphics g, int n,
          int x, int y, int z, int t) {
    int u, v;
    if (n == 1)
      g.drawLine (x, y, z, t);
    else {
      u = (x + z + t - y) / 2;
      v = (y + t - z + x) / 2;
      drawDragon (g, n - 1, x, y, u, v);
      drawDragon (g, n - 1, z, t, u, v);
    }
  }
}
```


Exécution d'une applette

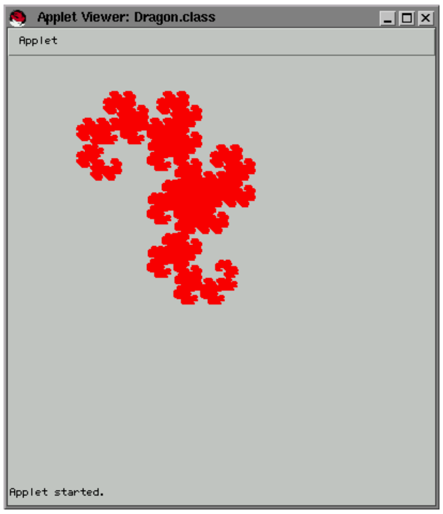
On crée un fichier `DessinDragon.html` contenant

```
<HTML>
<HEAD>
<TITLE> Courbe du dragon </TITLE>
</HEAD>

<BODY>
<APPLET CODE="Dragon.class" WIDTH=400
        HEIGHT=400> </APPLET>
</BODY>
</HTML>
```

On ouvre ensuite ce fichier sous Netscape ou directement par

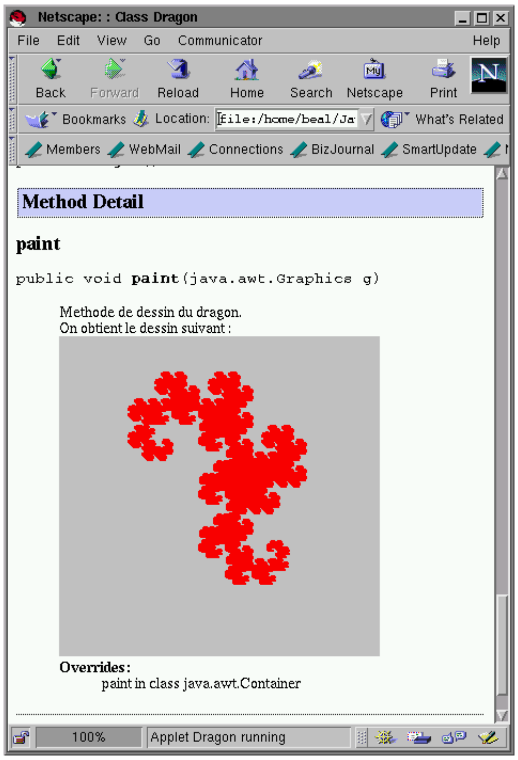
```
appletviewer DessinDragon.html
```

Le dragon



Le dragon dans la documentation

```
import java.awt.*;
import java.applet.*;
public class Dragon extends Applet {
 /**
  * Methode de dessin du dragon.
  * </BR>
  * On obtient le dessin suivant :
  * </BR>
  * <APPLET CODE="Dragon.class" WIDTH=300 HEIGHT=300> </APPLET>
  */
  public void paint(Graphics g) {
    g.setColor(Color.red);
    drawDragon(g, 20, 100, 100, 200, 200);
  }
  private void drawDragon (Graphics g, int n,
      int x, int y, int z, int t) {
    int u, v;
    if (n == 1)
      g.drawLine (x, y, z, t);
    else {
      u = (x + z + t - y) / 2;
      v = (y + t - z + x) / 2;
      drawDragon (g, n - 1, x, y, u, v);
      drawDragon (g, n - 1, z, t, u, v);
    }
  }
}
```

Le langage

Java (nom dérivé de Kawa) a vu le jour en 1995. Actuellement version 1.3 (aussi appelée Java 2 version 1.3). La version 1.4 est en beta.

Java

- est fortement typé,
- est orienté objet,
- est compilé–interprété,
- intègre des *thread* ou processus légers,
- est sans héritage multiple,
- n'offre pas de généricité (classes paramétrées comme les *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).

Plus de mille 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.

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

- 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 donnée *de classe* : par le nom de la classe.

  ```
  Math.cos()     Math.PI
  ```
- Méthode ou donnée d'un objet : par le nom de l'objet.

  ```
  Pile p;
  ...
  p.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.

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.

Exemple :

```
Point p;
```

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

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

L'évaluation de l'expression `new Point(4, 6)` retourne une référence à un objet de la classe `Point`. 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 incrementer(Point a) {
  a.x++; a.y++;
}
```

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

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 plateforme.
- 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
```


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.)

Expressions

Comme en C :

```
radians = (degres/180) * Math.PI // no comment
"Bonjour" + " Monde"  // String
(i != 0) && (i % 2 == 0)  // boolean
x = x +1
++i   i++   --j   j--
j += 2; // j = j+2
j += j; // j = j+j
max = (a > b) ? a : b;
(x % 2 == 1) ? 3*x+1 : x/2
```


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[] a; // 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` :

```
a = 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 < a.length; i++)
  System.out.print( a[i] );
```

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]`.

Matrices

Les tableaux sont dynamiques dans toutes les dimensions.

```
/**
 * Classe permettant de manipuler des matrices.
 * @author Beal
 * @author Berstel
 * @version 1.0
 */

class Matrice {
  int[][] m;

  /**
   * Cree une matrice nulle dont la
   * taille est donnee en parametre.
   * @param n taille de la matrice
   * @see Matrice#Matrice(int,int)
   */
  Matrice(int n) {
    m = new int [n][n];
    // de facon presque equivalente : this(n,0);
  }
```

```
/**
 * Cree une matrice dont la taille et le
 * contenu sont donnes en parametres.
 * @param n taille de la matrice
 * @param x valeur de chaque coefficient
 * @see Matrice#Matrice(int)
 */
Matrice(int n, int x) {
  m = new int [n][n];
  for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    m[i][j]=x;
}

/**
 * Methode de transposition d'une matrice
 */
void transposer() {
  int n = m.length;
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++) {
      int t = m[i][j];
      m[i][j] = m[j][i];
      m[j][i]=t;
  }
}
```

Noter la surcharge du constructeur.


```
  /**
   * Methode d'affichage d'une matrice
   */
  public String toString() {
    int n = m.length;
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++)
        sb.append(m[i][j] + " ");
      sb.append("\n");
    }
    return sb.toString();
  }

 /**
  * Methode principale
  * @param args arguments de la ligne de commande
  */
  public static void main(String[] args) {
    Matrice a = new Matrice(3,12);
    System.out.print(a);
    Matrice b = new Matrice(3);
    System.out.print(b);
  }
}
```

On obtient :

```
$ java Matrice
12 12 12
12 12 12
12 12 12
0 0 0
0 0 0
0 0 0
```


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
    }
  }
}
```

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 debut;
  Pixel fin;
}
```

Utilisation :

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

Plusieurs constructeurs pour la même classe :

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


Noter que

- dans le deuxième constructeur, on affecte à `debut` et `fin` 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.debut = null, s.fin = null
  s.debut = new Pixel(2,3);
  s.fin = 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 Point {
   int x, y;
   static Point origine = new Point(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());
  }
}
```


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
```


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`.
- La relation d'héritage est transitive.

```
class Base {
  private int p = 2;
  int x = 3, y = 5;
  public String toString(){
    return "B "+p+" "+x+" "+y;
  }
  int somme(){return x+y;}
}
class Der extends Base {
  int z=7;
  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.somme());
    System.out.println(d);
    System.out.println(d.somme());
  }
}
```

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, corps dérive d'anneau, matrice symétrique dérive de matrice.
- un *enrichissement* de la classe de base.
  Un segment coloré dérive d'un segment. Un espace vectoriel normé dérive d'un espace vectoriel. 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*.

Points épais : exemple de dérivation

```
class Point {
  private int x, y;
  Point (int x, int y) {
    this.x = x; this.y = y;
  }
  void translater(int dx, int dy) {
    this.x += dx; this.y += dy;
  }
  public String toString() {
    return(this.x + ", " + this.y);
  }
}

class PointEpais extends Point {
  int épaisseur;
  PointEpais(int x, int y, int e) {
    super(x, y);
    épaisseur = e;
  }
  public String toString() {
    return super.toString() + ", " + épaisseur;
  }
  void epaissir(int i) {
    épaisseur += i;
  }
  public static void main(String[] args) {
    PointEpais a = new PointEpais(3, 5, 1);
    System.out.println(a); // 3, 5, 1
    a.translater(5,-2); System.out.println(a); // 8, 3, 1
    a.epaissir(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`.

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 référence 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

```
class Point {
  private int x, y;
  Point (int x, int y) {
    this.x = x; this.y = y;
  }
  void translater(int dx, int dy) {
    this.x += dx; this.y += dy;
  }
  public String toString() {
    return(this.x + ", " + this.y);
  }
}
class Disque {
  Point centre;
  int rayon;
  Disque(int x, int y, int rayon) {
    centre = new Point(x,y);
    this.rayon = rayon;
  }
  public String toString() {
    return centre.toString() + " ," + rayon;
  }
  void translater(int dx, int dy) {
    centre.translater(dx, dy); // délégation
  }
}
class TestDisque {
  public static void main(String[] args) {
    Disque d = new Disque(3, 5, 1);  System.out.println(d); // 3, 5 ,1
    d.translater(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.

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 dynamiquement en fonction du type de l'objet à sa création. Ce mécanisme est appelé *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 {
     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

- 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.

Principe de base : *Une variable de type B peut contenir implicitement une référence à un objet de toute classe dérivée D de B.*
Exemple :

```
Point s = new PointEpais(5, 7, 1);
```

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.

```
Point s = new PointEpais(5, 7, 1);
PointEpais t;
t = s;            // erreur
t = (PointEpais) s;  // ok
```

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 dynamique (à l'exécution).

Intérêt du transtypage

Si une variable x de type B contient une référence à un objet d'une classe dérivée D de B, seuls les attributs et les méthodes dont le nom est connu dans la classe B sont accessibles par la variable x.
Si m est une telle méthode, l'appel $x.m()$ invoque la méthode m telle qu'elle est définie dans la classe de l'*objet* référencé par x. Ici, la méthode m telle que définie dans D : **polymorphisme**.
Le transtypage permet

- l'*encapsulation*, en rendant inaccessible les détails de D.
- l'*abstraction*, en utilisant des méthodes génériques qui sont spécialisées dans les classes dérivées.

Héritage : interfaces

Une *interface*

- 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.
C'est le point ultime de l'abstraction. Un style de programmation à encourager:

Program to Interfaces

Rectangles et ellipses

Une classe `Rectangle` comme avant :

```
class Rectangle {
  double largeur, hauteur;
  Rectangle(double largeur, double hauteur) {
    this.largeur = largeur; this.hauteur = hauteur;
  }
  double getAire() {return largeur*hauteur;}
  String toStringAire() {
    return "aire = "+getAire();
  }
}
```

et une classe `Ellipse` donnée par

```
class Ellipse {
  double largeur, hauteur;
  Ellipse(double largeur, double hauteur) {
    this.largeur = largeur; this.hauteur = hauteur;
  }
  double getAire(){
    return largeur*hauteur*Math.PI/4;}
  String toStringAire() {
    return "aire = "+ getAire();
  }
}
```

Ces classes ont une *abstraction commune*, qui

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

L'*interface* `Forme`

```
interface Forme {
  double getAire();
  String toStringAire();
}
```

La classe *abstraite* `AbstractForme` définit l'implémentation de `toStringAire`

```
abstract class AbstractForme implements Forme {
  double largeur, hauteur;
  AbstractForme(double largeur, double hauteur) {
    this.largeur = largeur;
    this.hauteur = hauteur;
  }
  public String toStringAire() {
    return "aire = " + getAire();
  }
}
```

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

```
class Rectangle extends AbstractForme{
  Rectangle(double largeur, double hauteur) {
    super(largeur, hauteur);
  }
  public double getAire() { return largeur*hauteur; }
}
```

et

```
class Ellipse extends AbstractForme {
  Ellipse(double largeur, double hauteur) {
    super(largeur, hauteur);
  }
  public double getAire() { return Math.PI*largeur*hauteur/4; }
}
```

On s'en sert par exemple dans :


```
Forme r = new Rectangle(6,10);
Forme c = new Ellipse(3,5);
System.out.println(r.toStringAire());
System.out.println(c.toStringAire());
```

ou dans :

```
Forme[] a = new Forme[5];
a[0] = new Rectangle(6,10);
System.out.println(a[0].toStringAire());
...
```


Instanceof

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

```
Forme f = new Rectangle(6,10);

if (f instanceof Rectangle) {...} // vrai
if (f instanceof Ellipse) {...} // faux
```

- notion de polymorphisme

Il ne faut pas abuser de `instanceof`.

Exemples d'interface

Matrices

```
interface Matrice { // Matrice d'entiers
  Matrice add(Matrice a);
  void setAt(int i, int j, int valeur);
  void transposer();
}
```

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

```
public abstract class AbstractMatrice implements Matrice
  int[][] m;
  AbstractMatrice(int n) {
    m = new int [n][n];
    // de facon presque equivalente : this(n,0);
  }
  AbstractMatrice(int n, int x) {
    m = new int [n][n];
    for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      m[i][j]=x;
  }
  public String toString() {
    int n = m.length;
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < n; i++) {
```

```
        for (int j = 0; j < n; j++)
          sb.append(m[i][j] + " ");
        sb.append("\n");
      }
      return sb.toString();
    }
  }
```

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

```
  public class GenMatrice extends AbstractMatrice{
    GenMatrice(int n) {
      super(n);
    }
    GenMatrice(int n, int x) {
      super(n, x);
    }
    public Matrice add(Matrice a) {
      int n = m.length;
      Matrice r = new GenMatrice(n);
      for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
          r.setAt(i, j, m[i][j] + a.m[i][j]);
      return r;
    }
    public void setAt(int i, int j, int valeur) {
        m[i][j] = valeur;
    }
    public void transposer() {
      int n = m.length;
      for (int i = 0; i < n; i++)
```



```
        for (int j = i+1; j < n; j++) {
          int t = m[i][j];
          m[i][j] = m[j][i];
          m[j][i]=t;
        }
      }
    }

  public class SymMatrice extends AbstractMatrice{
    SymMatrice(int n) {
      super(n);
    }
    SymMatrice(int n, int x) {
      super(n, x);
    }
    public Matrice add(Matrice a) {
      if (! (a instanceof SymMatrice))
        throw new UnsupportedOperationException();
      int n = m.length;
      SymMatrice sa = (SymMatrice) a;
      Matrice r = new SymMatrice(n);
      for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
          r.setAt(i, j, m[i][j] + sa.m[i][j]);
      return r;
    }
    public void setAt(int i, int j, int valeur) {
        m[i][j] = m[j][i] = valeur;
    }
    public void transposer() {}

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



```
      Matrice a = new SymMatrice(3,12);
      System.out.print(a);
      Matrice b = new SymMatrice(3);
      b.setAt(1,2,4);
      System.out.print(b);
      Matrice c = b.add(b);
      System.out.print(c);
    }
  }
```

Résultat :

```
12 12 12
12 12 12
12 12 12
0 0 0
0 0 4
0 4 0
0 0 0
0 0 8
0 8 0
```

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, implémentée par un tableau

```
public class Pile {
  static final int maxP=4;
  int hauteur = 0;
  int[] contenu = new int[maxP];

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

  public boolean isFull() {
    return hauteur == maxP;
  }

  public void push(int x) {
    contenu[hauteur++] = x;
  }

  public int top() {
    return contenu[hauteur-1];
  }

  public void pop() {
    --hauteur;
  }
}
```


Il y a *débordement* lorsque l'on fait

- `top` 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 une *exception*.

Une *exception* est un objet d'une classe qui étend la classe `Exception`. La classe `Exception` dérive de `Throwable`. La classe `Throwable` a une autre classe dérivée, la classe `Error`.
Pour les piles, on peut définir

```
class PileException extends Exception {}
```

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

```
void push(int x) throws PileException {
  if (isFull())
    throw new PileException("Pile pleine");
  contenu[hauteur++] = x;
}
```

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

```
...
Pile p = new Pile();
try {
  System.out.println("top = "+p.top());
} catch(PileException 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 (PileException 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 de la classe `RuntimeException` n'ont pas à être capturées.

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

```
interface Pile {
  boolean isEmpty ();
  boolean isFull();
  void push(int x) throws PileException;
  int top() throws PileException ;
  void pop() throws PileException;
}

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


Implémentation par tableau

```
public class TPile implements Pile {
  static final int maxP=4;
  private int hauteur = 0;
  private int[] contenu = new int[maxP];

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

  public boolean isFull() {
    return hauteur == maxP;
  }

  public void push(int x) throws PileException {
    if (isFull())
      throw new PileException("Pile pleine");
    contenu[hauteur++] = x;
  }

  public int top() throws PileException {
    if (isEmpty())
      throw new PileException("Pile vide");
    return contenu[hauteur-1];
  }

  public void pop()  throws PileException {
    if (isEmpty())
      throw new PileException("Pile vide");
    --hauteur;
  }
}
```


Implémentation par liste (classe interne liste)

```
public class LPile implements Pile {
  private Liste tete = null;
  class Liste {
    int cont;
    Liste suiv;
    Liste(int cont, Liste suiv) {
      this.cont = cont; this.suiv = suiv;
    }
  }

  public boolean isEmpty() {
    return tete == null;
  }

  public boolean isFull() {
    return false;
  }

  public void push(int x) throws PileException {
    tete = new Liste(x, tete);
  }

  public int top() throws PileException {
    if (isEmpty())
      throw new PileException("Pile vide");
    return tete.cont ;
  }

  public void pop()  throws PileException {
    if (isEmpty())
      throw new PileException("Pile vide");
    tete = tete.suiv;
  }
}
```


Usage:

```
public class TestPileImpl {
  public static void main(String[] args) {
    Pile p = new TPile(); //par table
    try {
      p.push(2); p.pop();
      p.pop(); // ca coince
      p.pop(); // jamais atteint
    }
    catch(PileException e) {
      System.out.println(e.getMessage());
      e.printStackTrace();
    }
    Pile q = new LPile();  //par liste
    try {
      q.push(2);  q.push(5);
      q.pop(); q.pop();
      System.out.println(q.top()); // ca coince
    }
    catch(PileException e) {
      System.out.println(e.getMessage());
      e.printStackTrace();
    }
  }
}
```

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

```
Chose createChose() {return new Chose(); }
```

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(Pile p) {
  try {
    p.push(2); p.push(5);
    p.pop(); p.pop();
    System.out.println(p.top());
  }
  catch(PileException e) {
    System.out.println(e.getMessage());
  }
}
```

utilisé avec

```
testVersion1(new PileT());
testVersion1(new PileL());
```

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

```
public static void testVersion2(boolean version) {
  Pile p;
  if (version)
    p = new PileT();
  else
    p = new PileL();
  ...
}
```


utilisé avec

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

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

```
public static Pile createPile(boolean version) {
  if (version) return new PileT();
  else return new PileL();
}

public static void testVersion3(boolean version) {
  Pile p;
  p = createPile(version);
  ...
}
```

utilisé avec

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

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

```
public static void testVersion4(Class classPile)
  throws IllegalAccessException, InstantiationException   {
  Pile p;
  p = (Pile) classPile.newInstance();
  ...
}
```

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

```
try {
  testVersion4(PileT.class);
  testVersion4(PileL.class);
}
catch(IllegalAccessException e) {}
catch(InstantiationException e) {}
catch(ClassNotFoundException  e) {}
```


Paquetage

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 MonApplette extends Applet` *non trouvée*

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

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

`import java.applet.*;`
`class MonApplette 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.

Programmation des listes

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 ***liste*** est

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

Ceci conduit à une interface pour les listes, avec deux interfaces, pour le Cons et la liste vide.

```
public interface Liste {
  int length();
}
public interface Cons extends Liste {
  Object getElem();
  void setElem(Object o);
  Liste getSuiv();
  void setSuiv(Liste tail);
}
public interface Nil extends Liste {}
```


L'implémentation se fait naturellement:

```
public class ConcreteCons implements Cons{
  private Object o;
  private Liste suiv;
  public ConcreteCons(Object o, Liste suiv){
    this.o = o;
    this.suiv = suiv;
  }
  public Object getElem(){
    return o;
  }
  public void setElem(Object o){
    this.o = o;
  }
  public Liste getSuiv(){
    return suiv;
  }
  public void setSuiv(Liste suiv){
    this.suiv = suiv;
  }
  public int length(){
    return suiv.length()+1;
  }
}

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


Usage:

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

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

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

avec bien sûr

```
    l = new ConcreteCons(new Integer(1),
          new ConcreteCons(new Integer(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 Liste getNil() {
    if (instance == null)
      instance = new ConcreteNil();
    return instance; }
  public int length(){ return 0; } // autres méthodes
}
```


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 fonction à nombre variable d'arguments
- de valeurs par défaut dans les fonctions
- de pointeurs (pas de pointeurs de fonction)
- de généricité (template)
- 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`) pour suppléer (mal) à l'absence de généricité
- 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.