# Programmation objet et JAVA


0. Bibliographie
1. Programmation objet
2. Le langage Java
3. Les classes fondamentales
4. La programmation concurrente
5. Les flots
6. Les composants AWT de Java
7. Introduction à Swing




# 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


1. Styles de programmation
2. Avantages du style objet
3. Un exemple
4. Héritage et composition
5. Exemple : disques et anneaux

## 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

1. Quelques exemples
2. Le langage
3. Structure d'un programme
4. Eléments de base
5. Classes et objets
6. Héritage : généralités
7. Héritage : exemples
8. Héritage : interfaces
9. Exceptions
10. Visibilité et paquetages
11. Créer une documentation
12. Programmation des listes
13. Java et C++



## 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();
  }
}
```

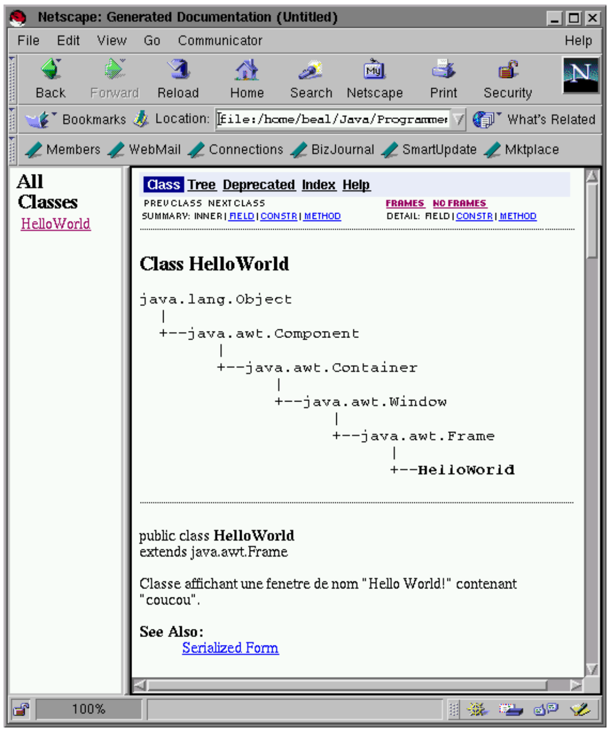
Création de la documentation HTML

```
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()
*/
```

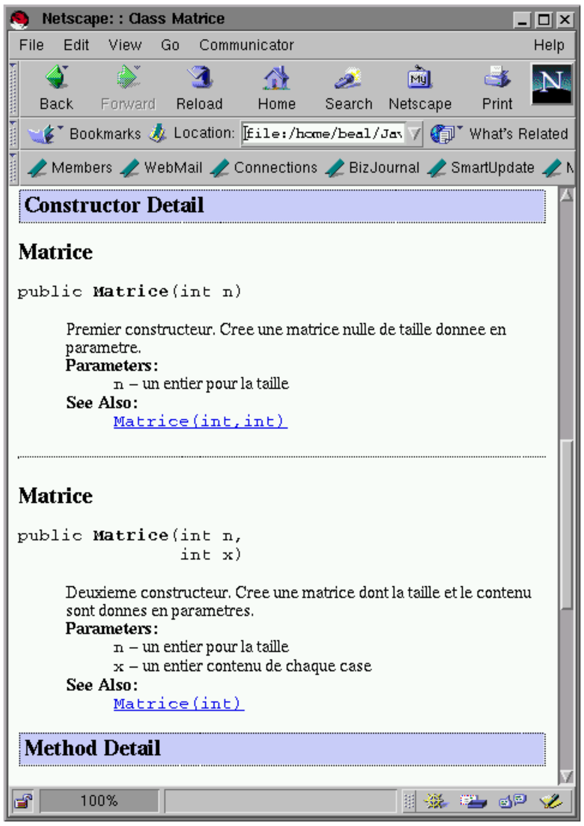


*Un exemple* :

```
/**
 * 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

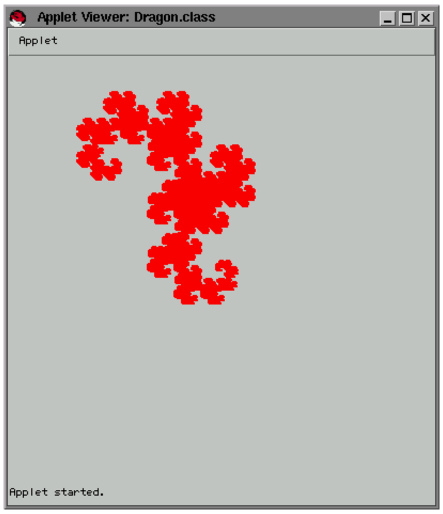
On crée un fichier `DessinDragon.html` contenant

```
<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

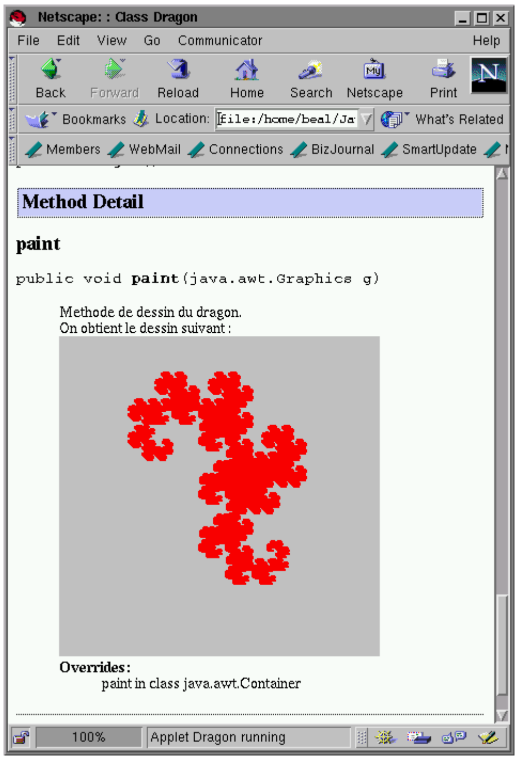





# Le dragon dans la documentation

```
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.



# 3

# Les classes fondamentales

1. Présentation des API
2. Les enveloppes des types de base
3. La classe `java.lang.Object`
   mère de toutes les classes
4. Les chaînes de caractères
5. Outils mathématiques
6. Ensembles structurés, itérateurs et comparateurs
7. 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) |

et de nombreuses autres.

## Les enveloppes des types de base

| | |
|---|---|
| `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. Par exemple `intValue()` renvoie un `int`. Les noms varient d'une classe à l'autre.
- 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.



## La classe `java.lang.Object`

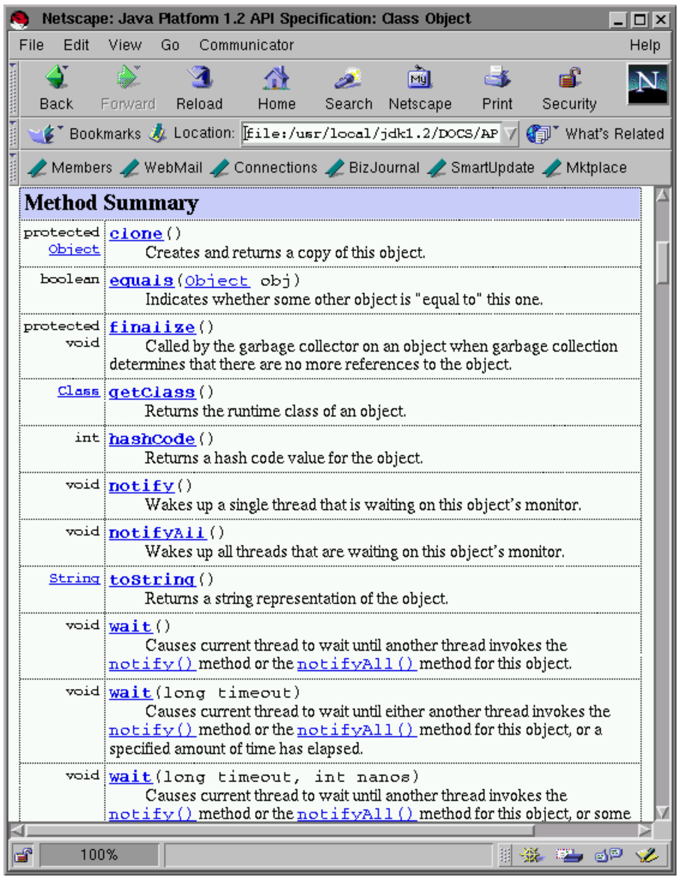
### Affichage d'un objet et hashcode

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
```

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






**Method Summary**

| | |
|---|---|
| protected Object | `clone()` Creates and returns a copy of this object. |
| boolean | `equals(Object obj)` Indicates whether some other object is "equal to" this one. |
| protected void | `finalize()` Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. |
| Class | `getClass()` Returns the runtime class of an object. |
| int | `hashCode()` Returns a hash code value for the object. |
| void | `notify()` Wakes up a single thread that is waiting on this object's monitor. |
| void | `notifyAll()` Wakes up all threads that are waiting on this object's monitor. |
| String | `toString()` Returns a string representation of the object. |
| void | `wait()` Causes current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object. |
| void | `wait(long timeout)` Causes current thread to wait until either another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified amount of time has elapsed. |
| void | `wait(long timeout, int nanos)` Causes current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object, or some |



## Le clonage

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

La classe `Object` contient une méthode `Object clone()`.

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

Exemple:

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

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

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

  public static void main(String[] args)
    throws CloneNotSupportedException
  {
    Point a = new Point(3,5);
    Point b = (Point) a.clone(); // méthode clone() de Object
    b.x = a.x + 1;
```

```
    System.out.println(a); // 3 5
    System.out.println(b); // 4 5
  }
}
```

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

### Exemple de clonage

Le clonage par défaut est superficiel.

```
public class Pile implements Cloneable {
   int hauteur;
   int[] contenu;

   public Pile (int maxP) {
     hauteur = 0;
     contenu = new int[maxP];
   }

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

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

  public Object clone() throws CloneNotSupportedException {
    Pile nouvelObjet = (Pile) super.clone();
    nouvelObjet.contenu = (int[]) contenu.clone();
    return nouvelObjet;
  }
}
```



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

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

```
public class DPile extends Pile {
  int cout;

  public DPile (int maxP) {
    super(maxP);
    cout = 0;
  }

  public Object clone() throws CloneNotSupportedException   {
    DPile nouvelObjet = (DPile) super.clone();
    nouvelObjet.cout = cout;
    return nouvelObjet;
  }
}
```



### Égalité entre objets

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

```
class Rectangle extends Forme{
  int largeur, hauteur;
  Rectangle(int largeur, int hauteur) {
    this.largeur = largeur;
    this.hauteur = hauteur;
  }
  public boolean equals(Object arg) {
    if (!(arg instanceof Rectangle))
      return false;
    Rectangle rarg = (Rectangle)arg;
    return (largeur == rarg.largeur)
      && (hauteur == rarg.hauteur);
  }
```

Remarquer que l'argument 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 `Forme` qui référence un `Rectangle`. La comparaison entre les deux rectangles serait alors incorrecte.



## Les chaînes de caractères

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

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

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

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

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

et inversement :

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

Conversion d'un entier en chaîne de caractère :

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

et inversement :

```
int i = Integer.valueOf("12").intValue(); // 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 méthode `trim()` supprime tous les espaces blancs :

```
String s = "   toto     ";
s = s.trim();
```

*La classe* `java.lang.StringBuffer`

- permet de créér des buffers de caractères de taille extensible.

```
StringBuffer sb = new StringBuffer("le");
sb.append(" petit ");
sb.append("prince");
```

On récupère un `String` à partir d'un `StringBuffer` (sans recopie) :

```
String mot = sb.toString();
```

Ces classes sont sécurisées au niveau des *thread* (voir plus tard).



*La classe* `java.util.StringTokenizer` permet d'effectuer de l'analyse lexicale simple.

```
String texte = "le petit prince";
StringTokenizer st = new StringTokenizer(texte);
while (st.hasMoreTokens())
   { String mot = st.nextTokens();}
```

La classe `StringTokenizer` implémente l'interface `java.util.Enumeration`.
Les délimiteurs par défaut sont les espaces, retours chariot et tabulations. On peut spécifier les délimiteurs :

```
String texte = "http://www-igm.univ-mlv.fr/~beal";
StringTokenizer st = new StringTokenizer(texte,"/:~");
if (st.countTokens() < 2) ...  // mauvaise URL
String protocole = st.nextToken();
String host = st.nextToken();
String login = st.nextToken();
```



## Outils mathématiques

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

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

Exemple : `int maximum = Math.max(3,4);`
Exemple : Tirer au hasard un entier entre 100 et 1000 (les deux compris).

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

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

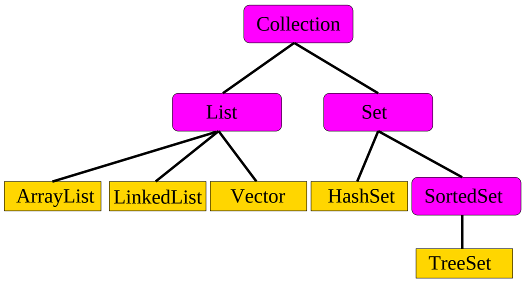
```
try { int i = 1/0;}
catch (ArithmeticException e) {...}
```

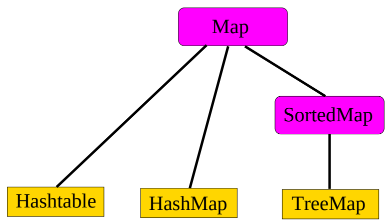
- Une instruction arithmétique sur les flottants (`double`) ne lève pas d'exception. Une expression flottante peut prendre trois valeurs particulières :

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



## Ensembles structurés, itérateurs et comparateurs



Regardons par exemple :

- `java.util.ArrayList`

```
$ javap java.util.ArrayList
public class java.util.ArrayList
extends java.util.AbstractList implements java.util.List,
java.lang.Cloneable, java.io.Serializable {
    public java.util.ArrayList();
    public java.util.ArrayList(int);
    public void add(int, java.lang.Object);
    public boolean add(java.lang.Object);
    public void clear();
    public java.lang.Object clone();
    public boolean contains(java.lang.Object);
    public java.lang.Object get(int);
    public int indexOf(java.lang.Object);
    public boolean isEmpty();
    public java.lang.Object remove(int);
    public java.lang.Object set(int, java.lang.Object);
    public int size();
}
```

- `java.util.Iterator`

```
$ javap java.util.Iterator
public interface java.util.Iterator {
    public abstract boolean hasNext();
    public abstract java.lang.Object next();
    public abstract void remove();
}
```



## Vue d'ensemble

Deux interfaces

- `Collection` pour les ensembles d'objets,
- `Map` pour les tables, c'est-à-dire des ensembles de couples (clé, valeur), où la clé et la valeur sont des objets.

Des itérateurs sur les collections

- `Iterator` interface des itérateurs,
- `ListIterator` itérateur sur les séquences.
- `Enumeration` ancienne forme des itérateurs.

De plus, deux classes d'utilitaires

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

Les opérations principales sur une collection

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

Comme les éléments sont des objets, ne pas écrire `c.add(2)` mais `c.add(new Integer(2))`.



Sous-interfaces spécialisées de `Collection`

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

Opérations ensemblistes sur les collections

- `boolean containsAll(Collection c)` pour tester l'inclusion.
- `boolean addAll(Collection 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` (recommandée, par tableau), et `LinkedList` (par liste doublement chaînée) implémentent `List`.
- `Vector` est une vieille classe (JDK 1.0) "relookée" qui implémente aussi `List`. Elle a des méthodes personnelles.
- `HashSet` (recommandée) implémente `Set`.
- `TreeSet` implémente `SortedSet`.

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

Discipline d'abstraction:

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

Exemple

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

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

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

class TestSet {
  public static void main(String[] args) {
    Set s = new HashSet();
    for (int i = 0; i < args.length; i++)
      if (!s.add(args[i]))
        System.out.println("Déjà vu : " + args[i]);
    System.out.println(s.size() + " distincts : " + s);
  }
}
```

```
$ java TestSet a b c a b d
Déjà vu : a
Déjà vu : b
4 distincts : [d, c, b, a]
```



Itérateurs

L'interface `Iterator` définit les itérateurs.

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

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

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

- `boolean hasNext()` qui teste si le parcours contient encore des éléments;
- `Object 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`.

L'interface `java.util.Enumeration` a deux méthodes

- `boolean hasMoreElements()` qui teste si l'énumération contient encore des éléments;
- `Object nextElements()` qui retourne l'élément suivant, si un tel élément existe (et une exception sinon).

```
import java.util.*;

class Personne {
  String nom;
  int age;
  public Personne(String nom, int age) {
    this.nom = nom; this.age = age;
  }
  public String toString() { return "Nom: "+nom+", age: "+age; }
```



```
}

class TestHashSet {
  public static void printAll(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
      System.out.println(i.next());
  }
  public static void main(String[] args) {
    Set s = new HashSet();
    s.add(new Personne("Pierre", 23));
    s.add(new Personne("Anne", 20));
    s.add("Université");
    s.add("Marne-la-Vallée");
    printAll(s);
    Set t = (Set) ((HashSet) s).clone();
    System.out.println(s.size());
    printAll(t);
    Iterator i = t.iterator();
    while(i.hasNext())
      if (i.next() instanceof Personne) i.remove();
    printAll(t);
  }
}
```

Avec les résultats

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

Observer le désordre.



Détails sur les itérateurs.

- la méthode `iterator()` de la collection positionne l'itérateur au “début”,
- la méthode `hasNext()` teste si l'on peut progresser,
- la méthode `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()`.

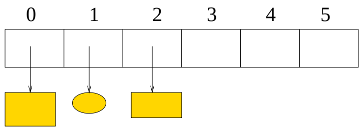
```
|A B C     iterator(), hasNext() = true
 A|B C     next() = A, hasNext() = true
 A B|C     next() = B, hasNext() = true
 A B C|    next() = C, hasNext() = false
```

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

## Exemple de tableau

On désire créér un tableau des références sur des objets de type `Forme` qui peuvent être `Rectangle` ou `Ellipse` (voir chapitre 1).

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

abstract class Forme {
  abstract double getAire(); // a définir
  String toStringAire() {
    return "aire = " + getAire();
  } // commune
}
class Rectangle extends Forme{ ... }
class Ellipse extends Forme { ... }

class TestForme {
  public static void main(String[] args) {
    Forme r1 = new Rectangle(6,10);
    Forme r2 = new Rectangle(5,10);
    Forme e = new Ellipse(3,5);
    List liste = new ArrayList();
    liste.add(r1);
    liste.add(r2);
    liste.add(1,e);  // on a r1, e, r2
    for (Iterator it = liste.iterator(); it.hasNext();)
      System.out.println( ((Forme) it.next()).toStringAire());
  }
}
```






## Itérer sur les listes

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

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

```
import java.util.*;

class TestLinkedList {
  public static void printAll(Collection c) { ... }
  public static void main(String[] args) {
    List a = new LinkedList();
    a.add("A");
    a.add("B");
    a.add("C");
    printAll(a);  // A B C
    ListIterator 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 `Comparable`.

L'interface `Comparable` déclare une méthode `int compareTo(Object o)` telle que `a.compareTo(b)` est

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

Exemple : comparaisons de "noms".

```
import java.util.*;

class Nom implements Comparable {
  private String nom;
  private int age;

  public Nom(String nom, int age) {
    this.nom = nom; this.age = age;
  }
  public int compareTo(Object o) {
    Nom n = (Nom) o;
    int comp = nom.compareTo(n.nom);
    return (comp !=0) ? comp : n.age - age;
  }
  public String toString() { return nom + " : " + age; }
}
```



```
class TestComparaison {
  public static void main(String[] args) {
    Collection c = new TreeSet();
    c.add(new Nom("Paul", 21));
    c.add(new Nom("Paul", 25));
    c.add(new Nom("Anne", 25));
    for (Iterator i = c.iterator(); i.hasNext(); )
      System.out.println(i.next());
  }
}
```

avec le résultat

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

## Comparateur

Un *comparateur* est un objet qui permet la comparaison. En Java, l'interface `Comparator` déclare une méthode
`int compare(Object a, Object b)`
On se sert d'un comparateur

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

Exemple de deux comparateurs de noms:



```
class NomComparator implements Comparator {
  public int compare(Object oa, Object ob) {
    Nom a = (Nom) oa;
    Nom b = (Nom) ob;
    int comp = a.nom().compareTo(b.nom());
    if (comp == 0)
      comp = a.age() -b.age();
    return comp;
  }
}
class AgeComparator implements Comparator {
  public int compare(Object oa, Object ob) {
    Nom a = (Nom) oa;
    Nom b = (Nom) ob;
    int comp = b.age() - a.age();
    if (comp == 0)
      comp = a.nom().compareTo(b.nom());
    return comp;
  }
}
```

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

```
public static void main(String[] args) {
  List c = new ArrayList();
  c.add(new Nom("Paul", 21));
  c.add(new Nom("Paul", 25));
  c.add(new Nom("Anne", 25));
  printAll(c);
  Collections.sort(c, new NomComparator());
  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
Anne : 25  Paul : 25  Paul : 21  // ordre sur ages
```



## Tables

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

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

La sous-interface `SortedMap` spécifie les tables dont l'ensemble des ***clés*** est ordonné.
Implémentation d'une table
Pour les tables

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

La classe `TreeMap` implémente les opérations avec des arbres rouge-noir.
Un `TreeMap` stocke les références de ces objets de telle sorte que les opérations suivantes s'exécutent en temps $O(\log(n))$ :



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

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

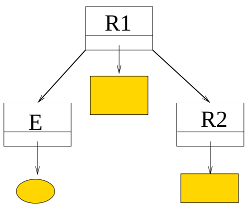
## Formes nommées

On associe un nom à chaque forme. Le nom est la *clé* de la forme.
L'interface `Forme`, et les classes `Rectangle` et `Ellipse` sont comme d'habitude.

```
import java.util.*;

public class FormeMapTest {
  public static void main(String[] args) {
    Forme r1 = new Rectangle(6,10);
    Forme r2 = new Rectangle(5,10);
    Forme e = new Ellipse(3,5);
    TreeMap arbre  = new TreeMap();
    arbre.put("R2",r2);
    arbre.put("R1",r1);
    arbre.put("E",e);
    System.out.println( ((Forme)arbre.get("R1")).toStringAire());
  }
}
```

On obtient :

```
$ java TestForme
aire = 60.0
```






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

La classe `java.util.TreeMap` possède un constructeur qui permet de changer le comparateur :

`public java.util.TreeMap(java.util.Comparator);`

Le programme devient :

```
import java.util.*;
public class OppositeComparator implements Comparator {
  public int compare(Object o1, Object o2) {
    return - ((Comparable) o1).compareTo(o2);  //ordre inverse
  }
}
```

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

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



## Itérer dans les tables

Les tables n'ont pas d'itérateurs.
Trois méthodes permettent de *voir* une table comme un ensemble

- `keySet()` retourne l'ensemble (`Set`) des clés;
- `values()` retourne la collection des valeurs associées aux clés;
- `entrySet()` retourne l'ensemble des couples (clé, valeur);

```
Map m = ...;
Set clés = m.keySet();
Set couples = m.entrySet();
Collection valeurs = m.values();
```

On peut ensuite itérer sur ces ensembles:

```
for (Iterator i = clés.iterator(); i.hasNext(); )
  System.out.println(i.next());

for (Iterator i = valeurs.iterator(); i.hasNext(); )
  System.out.println(i.next());

for (Iterator i = couples.iterator(); i.hasNext(); ) {
  Map.Entry e = (Map.Entry) i.next();
  System.out.println(e.getKey() + " -> " + e.getValue());
```



## 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 extends TreeMap {
  public void put(int page, String mot) {
    Set numeros = (Set) get(mot);
    if (numeros == null) {
      numeros = new TreeSet();
      put(mot, numeros); // la ‘‘vraie’’ méthode put
    }
    numeros.add(new Integer(page));
  }
  public void print() {
    Set clés = keySet();
    for (Iterator i = clés.iterator(); i.hasNext(); ) {
      String c = (String)i.next();
      System.out.println(c + " " + get(c));
    }
  }
}

class TestIndex {
  public static Index makeIndex() {
    Index index = new Index();
    index.put(22,"Java");
    index.put(23,"Itérateur");
    index.put(25,"Java");
    index.put(25,"Map");
    index.put(25,"Java");
    index.put(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 a;
...
Collections.sort(a);
```

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

```
List source = ...;
List 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 boules = new ArrayList(49);
    for (int i=0; i < 49; i++)
      boules.add(new Integer(i));
    Collections.shuffle(boules); // mélange
    List tirage = boules.subList(0,6); // les 6 premières
    Collections.sort(tirage); // tri
    System.out.println(tirage); // 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]
```



## Introspection

La classe `java.lang.Class` :

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

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

```
String s = "toto";
Class c = s.getClass();
interface I {}
Class c1 = I.class;
Class c2 = Class.forName("I");
Class c3 = float.class;
```

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

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

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

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

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

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

On obtient :

```
> java Test
Thu Sep 09 14:54:09 CEST 1999
Thu Sep 09 14:54:09 CEST 1999
Thu Sep 09 14:54:09 CEST 1999
```



## Le chargement de classes

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

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

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



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

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

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

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

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

La signature de `defineClass()` est :

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



```
import java.io.*;

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

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

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

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

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

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



Dans l'exemple précédent,

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

On obtient :

```
> java VerboseClassLoader
Chargement de A
Chargement de B
Chargement de java.lang.Object
Chargement de java.lang.Throwable
Chargement de java.lang.System
Chargement de java.io.PrintStream
nouveau A()
Chargement de D
Dans VerboseClassLoader : o n'est pas instance de A
VerboseClassLoader@4acca8a
nouveau A()
sun.misc.Launcher$AppClassLoader@1714ca8a
```



# 4

# La programmation concurrente

1. Programmation concurrente
2. Processus légers
3. Les `Thread`
4. Exclusion mutuelle
5. Synchronisation



## Programmation concurrente

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

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

## Processus légers

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

S'il y a plusieurs processus légers :

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

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

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

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



## La classe `java.lang.Thread`

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

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

On obtient à l'exécution :

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

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



## Démarrage et terminaison

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

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

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

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

La classe `Thread` possède sept constructeurs qui spécifient :

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



## Le lapin et la tortue (première version)

Classe des lapins

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

Classe des tortues

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

Mise en place

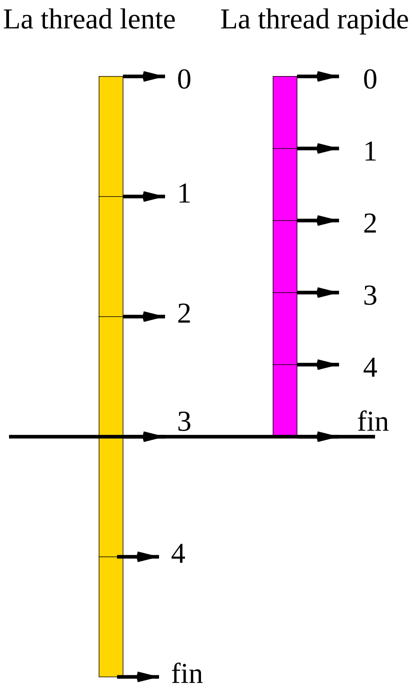
```
public class MesThread{
  public static void main(String[] args){
    Runnable tortue = new Tortue();
     Runnable lapin = new Lapin();

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

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

On obtient :

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

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








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

Classe des lapins étend Thread :

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

Classe des tortues étend Thread :

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

Mise en place

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

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



**Terminaison d'un processus léger**

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

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

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

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

*Priorités d'accès au processeur*

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

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

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

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

## Exclusion mutuelle

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

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

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

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

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

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



## Exemple : tableau

```
public class Tableau{
  private int[] tableau;

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

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

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



## Exemple : variante

```
public class Tableau{
  private int[] tableau;

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

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

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



## Sûreté et vivacité

Quelques notions :

- *sûreté (safety)* rien de faux peut se produire. L'exclusion mutuelle règle le problème de l'accès concurrent en écriture.
- *vivacité (liveness)* tout processus peut s'exécuter.

Les diverses facettes de la non vivacité :

- famine (contention): un processus léger est empêché de s'exécuter parce que un processus plus prioritaire accapare le processeur;
- endormissement (dormancy) : un processus léger est suspendu et jamais réveillé;
- terminaison prématurée;
- interblocage (deadlock) : plusieurs processus légers s'attendent mutuellement avant de continuer.

## Synchronisation entre processus légers

Java propose deux mécanismes :

- attente/notification avec `wait()` et `notify()`
  `wait()` appelé sur un objet suspend le processus léger courant qui attend une notification d'un autre processus via le moniteur de l'objet;
  `notify()` appelé sur un objet libère un processus léger en attente par `wait()` sur le moniteur du même objet.
- attente de terminaison avec `join()`
  `join()` est appelé sur l'objet de contrôle d'un processus léger dont la terminaison est attendue; le processus courant est alors interrompu jusqu'à la fin de celui-ci.

Les méthodes `wait()`, `join()` et `sleep()` peuvent être interrompues (et leur processus est alors débloqué). La méthode bloquante lève une exception `InterruptedException` qui peut être captée.



### Exemple : les tourneurs et le compteur

Cinq processus légers (les `Tourneur`) veulent faire tourner un compteur (le `Compteur`) qui compte modulo 5.
Voici le compteur:

```
public class Compteur {
  private int max; // 5 dans l'exemple
  private int count = 0; // initialisation, importante !
  public Compteur (int max) {
    this.max = max;
  }
  public int getMax(){
    return max;
  }
  public int getValue(){
    return count;
  }
  public synchronized void increment(){
    count = (count +1) % max ; // ce qu'un tourneur veut faire
  }
}
```

La règle du jeu : un Tourneur ne peut faire tourner le compteur que lorsqu'il est égal à son numéro.



Voici la classe des tourneurs.

```
public class Tourneur extends Thread{
  private Compteur c;  // le compteur
  private int numero;  // numéro du tourneur
  public Tourneur(int numero, Compteur c){
    System.out.println("Tourneur "+ numero + " créé.");
    this.numero = numero;
    this.c = c;
    if (numero + 1 < c.getMax()) {
      new Tourneur(numero + 1, c);
    }
    System.out.println("Tourneur "+ numero + " démarre.");
    start();
  }

  public void run(){ ... }

  public static void main(String[] args) {
    Compteur c = new Compteur(5);
    new Tourneur(0, c);
  }
}
```

Début d'exécution

```
Tourneur 0 créé.
Tourneur 1 créé.
Tourneur 2 créé.
Tourneur 3 créé.
Tourneur 4 créé.
Tourneur 4 démarre.
...
```



```
  public void run()
    try {
      for (int etape = 0 ; ; etape++) {
        System.out.println("Tourneur "+ numero
          + " in étape "+ etape);
        synchronized(c){
          while (numero != c.getValue())
            c.wait(); // suspend this
          System.out.println("Tourneur "+ numero
            + " out étape "+ etape);
          c.increment();
          c.notifyAll();  // libère les threads supendus
        }
      }
    } catch (InterruptedException e) {}
  }
```

Et le résultat:

```
...
Tourneur 4 démarre.
Tourneur 4 in étape 0
Tourneur 3 démarre.
Tourneur 3 in étape 0
Tourneur 2 démarre.
Tourneur 2 in étape 0
Tourneur 1 démarre.
Tourneur 1 in étape 0
Tourneur 0 démarre.
Tourneur 0 in étape 0
Tourneur 0 out étape 0
Tourneur 1 out étape 0
Tourneur 2 out étape 0
Tourneur 3 out étape 0
Tourneur 4 out étape 0
Tourneur 4 in étape 1
Tourneur 3 in étape 1
...
```

## Maître et esclave

Un esclave "travaille"

```
public class Esclave implements Runnable {
  private int result;
  public int getResult(){ return result; }
  public int durTravail(){ return 0; }
  public void run(){ result = durTravail(); }
}
```

Le maître fait travailler l'esclave, et attend, par `join`, la fin du processus esclave.

```
public class Maitre implements Runnable{
  public void run(){
    Esclave e = new Esclave();
    Thread esclave = new Thread(e);
    esclave.start();
    // fait autre chose
    try {
      esclave.join(); // attente de fin du run
    } catch (InterruptedException ex){}
    int result = e.getResult();
    System.out.println(result);
  }
}
```

Mise en place:

```
public class TestMaitre{
  public static void main(String[] args) {
    Maitre m = new Maitre();
    Thread maitre = new Thread(m);
    maitre.start();
  }
}
```



## Variables locales à un processus léger

On peut simuler des variables locales à chaque processus léger. Pour cela :

- on crée un objet de la classe `ThreadLocal`;
- on y accède par `Object get()`;
- on le modifie par `void set(Object o)`.

*Exemple*

```
public class MaCible implements Runnable {
    public ThreadLocal v = new ThreadLocal();
    public void run() {
      v.set(new Double(Math.random()));
      System.out.println(v.get());
    }
    public static void main(String[] args){
      MaCible c = new MaCible();
      Thread t1 = new Thread(c);
      Thread t2 = new Thread(c);
      t1.start();
      t2.start();
    }
}
```

On obtient :

```
0.8955847189505597
0.43636788900311063
```



# 5

## Les flots

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



## Généralités

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

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

*Pipeline ou filtrage*

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

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

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

Les classes sont toutes dans le paquetage `java.io`.
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.

## 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
   PrintWriter
   StringWriter
```



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

## Lire un octet

```
import java.io.*;

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

On obtient :

```
$$ java Lire
a
97
```

## Lire des octets

```
public class Lire {
  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 Lire
a é
97 32 233 10
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 octet 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 bufériser 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 `LineNumberReader`

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

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

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

### 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 LireMulti {
  public static void lire() 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 TestLireMulti{
  public static void main(String[] args) throws IOException{
    LireMulti.lire();
  }
}
```



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

$$ java TestLireMulti < 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.*;

public class InfoFichier{
  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());
  }
}
```

On obtient :

```
monge : > ls -l toto
-rw-r--r-- 1 beal institut 488 Sep 21 12:13 toto
monge : > java InfoFichier toto
toto
false
true
true
488
```

## 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 bloc.

```
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.

```
public String readFile(String f) throws IOException {
  FileReader fileIn = new FileReader(nom);
  BufferedReader in = new BufferedReader(fileIn);

  StringBuffer s = new StringBuffer();
  String line;
  while ((line = in.readLine()) != null)
    s.append(line + "\n");
  fileIn.close();
  return s.toString();
}
```



## 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 Point implements Serializable {
  private int x, y;
  public Point(int xx,int yy){ x = xx; y = yy; }
  public String toString(){ return "(" + x + "," + y + ")"; }

  public void sauvePoint(String nom) throws Exception {
    File f = new File(nom);
    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));
    Point oBis = (Point) 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{
    Point o = new Point(1,2);
    o.sauvePoint(args[0]);
  }
}
```

On obtient :

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



## Redéfinir l'objet de sauvegarde

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

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

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

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

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



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

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

On obtient :

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



# 6

## Les composants AWT de Java

(Abstract Windowing Toolkit)

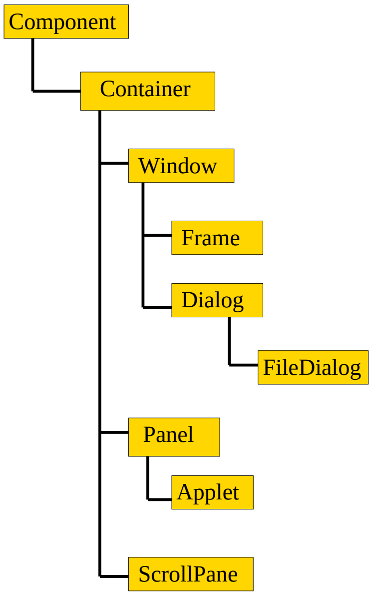
1. Composants et conteneurs
2. Événements
3. Dessins



### Composants et conteneurs

- Un *composant* est un objet de base de l'interface utilisateur de Java. C'est un objet d'un classe dérivée de la classe `java.awt.Component` (exemple : fenêtres, boutons, zones de dessin, menus, barres de défilement ...);
- Un composant est en principe inséré dans un *conteneur*. C'est un objet de la classe `java.awt.Container` qui regroupe des composants.
- Un *gestionnaire de placement* (Layout Manayer) gère la géométrie des composants. Il indique comment sont disposés les composants dans un conteneur.

## Différents types de conteneurs






## Gestionnaires de placement

Les gestionnaires de placement sont des objets de classes implémentant l'interface `LayoutManager`. Les principales sont :

- `FlowLayout`;
- `GridLayout`;
- `BorderLayout`;
- `BoxLayout`.

Chaque conteneur a un gestionnaire de placement par défaut. On peut le modifier par la méthode `setLayout()` de la classe `Container` (Ex : `setLayout(new FlowLayout());`).
Lorsque des objets sont ajoutés dans un conteneur, la mise à jour peut se demander par la méthode `validate()` de la classe `Component`.



## Événements

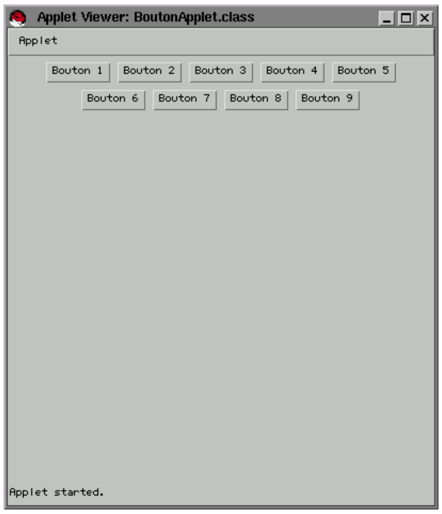
Les programmes à interfaces graphiques sont pilotés par des évènements. Un thread spéciale (`EventDispatchThread`) est créée par la machine virtuelle, en charge de la boucle de lecture et de distribution des évènements.
Un *événement* correspond à une action simple (enfoncement de la souris, pression d'une touche, ...) : on dit qu'il s'agit d'un *événement élémentaire*.
Un *événement sémantique* consiste en une séquence d'événements élémentaires (clic de souris, ...) synthétisés en un évènement unique.

### Le modèle émetteur-auditeur

- Un événement est émis (*fired*) par un composant;
- Un événement est transmis aux *auditeurs* enregistrés.
- Les auditeurs exécutent des méthodes en fonction de l'événement reçu.
- Plusieurs auditeurs peuvent être enregistrés pour un événement.
- Il existent différentes classes d'auditeurs (*listeners*) qui ont leurs propres méthodes. Ces classes sont des interfaces à implémenter.



## Créer des boutons en cliquant sur le premier



```
import java.awt.*;
import java.awt.event.*;

class BoutonNumerote extends Button {
  static int numero = 1;

  BoutonNumerote() {
    setLabel("Bouton "+ numero);
    numero++;
  }
}

public class BoutonFrame extends Frame {

  public BoutonFrame() {
    setTitle("Encore des boutons");
    Button b = new BoutonNumerote();
    add(b);
    b.addActionListener(new BoutonListener());
    setSize(200,200);
    setVisible(true);
  }

  public static void main(String[] args) {
    new BoutonFrame();
  }

  class BoutonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      add(new BoutonNumerote());
      validate();
    }
  }
}
```



## Version applette

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

class BoutonNumerote extends Button {
  static int numero = 1;

  BoutonNumerote() {
    setLabel("Bouton "+ numero);
    numero++;
  }
}

public class BoutonApplet extends Applet {

  public void init() {
    BoutonNumerote b = new BoutonNumerote();
    add(b);
    b.addActionListener(new BoutonListener());
  }

  class BoutonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      add(new BoutonNumerote());
      validate();
    }
  }
}
```

*Remarque :* La classe `BoutonListener` est une *classe interne* (*inner class*) à `BoutonApplet`.

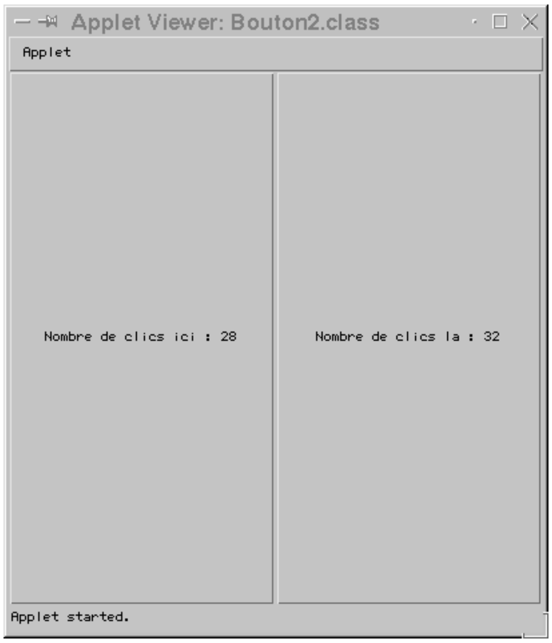


## Version plus compacte

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

class BoutonNumerote extends Button {
  static int numero = 1;

  BoutonNumerote() {
    super("Bouton "+ numero);
    numero++;
  }
}

public class BoutonApplet extends Applet
  implements ActionListener{

  public void init() {
    BoutonNumerote b = new BoutonNumerote();
    add(b);
    b.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    add(new BoutonNumerote());
    validate();
  }
}
```



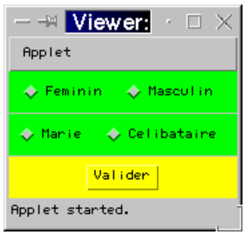
## Compter les clics sur des boutons



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

class BoutonCompteur extends Button {
  int compteur = 0;
  String nom;

  BoutonCompteur(String s) {
    super(s);
    nom = s;
  }
}


public class Bouton2 extends Applet implements ActionListener{
  BoutonCompteur b1,b2;

  public void init() {
    setLayout(new GridLayout(1,2,1,1));
    b1 = new BoutonCompteur("Nombre de clics ici : ");
    b1.addActionListener(this);
    b2 = new BoutonCompteur("Nombre de clics la : ");
    b2.addActionListener(this);
    add(b1);
    add(b2);
  }

  public void actionPerformed(ActionEvent e) {
    BoutonCompteur b = (BoutonCompteur) e.getSource();
    b.compteur++;
    b.setLabel(b.nom+b.compteur);
  }
}
```



## Cases à cocher




Sortie :

```
monge : >
Feminin
Celibataire
```



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

public class Cases extends Applet implements ActionListener{
  Panel sexePanel,mariPanel,validationPanel;
  CheckboxGroup sexeGroup,mariGroup;
  public void init() {
    setLayout(new GridLayout(3,1,1,1));
    sexePanel = new Panel();
    sexePanel.setBackground(Color.green);
    sexeGroup = new CheckboxGroup();
    sexePanel.add( new Checkbox("Feminin",sexeGroup,false));
    sexePanel.add( new Checkbox("Masculin",sexeGroup,false));

    mariPanel = new Panel();
    mariPanel.setBackground(Color.green);
    mariGroup = new CheckboxGroup();
    mariPanel.add( new Checkbox("Marie",mariGroup,false));
    mariPanel.add( new Checkbox("Celibataire",mariGroup,false));

    validationPanel = new Panel();
    validationPanel.setBackground(Color.yellow);
    Button b = new Button("Valider");
    b.addActionListener(this);
    validationPanel.add(b);
    add(sexePanel);
    add(mariPanel);
    add(validationPanel);
    setSize(getPreferredSize());
  }
  public void actionPerformed(ActionEvent e) {
    Checkbox c = sexeGroup.getSelectedCheckbox();
    Checkbox d = mariGroup.getSelectedCheckbox();
    if (c != null && d !=null)
      System.out.println(c.getLabel() + "\n" + d.getLabel());
  }
}
```



## Dessins

L'outil de dessin est le *contexte graphique*, un objet de la classe abstraite `Graphics`. Il encapsule des informations nécessaires concernant une zone de dessin comme :

- l'objet composant sur lequel on va dessiner;
- une translation d'origine;
- le rectangle de découpe;
- la couleur courante;
- la police de caractère courante;
- le mode opératoire logique de dessin (XOR ou Paint);
- éventuellement la couleur du XOR.

Le contexte graphique ne représente pas le dessin lui-même. On obtient un contexte graphique :

- *implicitement*, dans une méthode `paint()` ou `update()` : le contexte graphique construit est passé en paramètre;
- *explicitement*, en copiant un contexte graphique déjà existant;
- *explicitement*, en faisant un `getGraphics()` dans un composant ou une image.

L'acquisition explicite traduit une mauvaise utilisation de `paint()` ou `update()`. Il faut ensuite libérer le contexte graphique explicitement par `dispose()`.

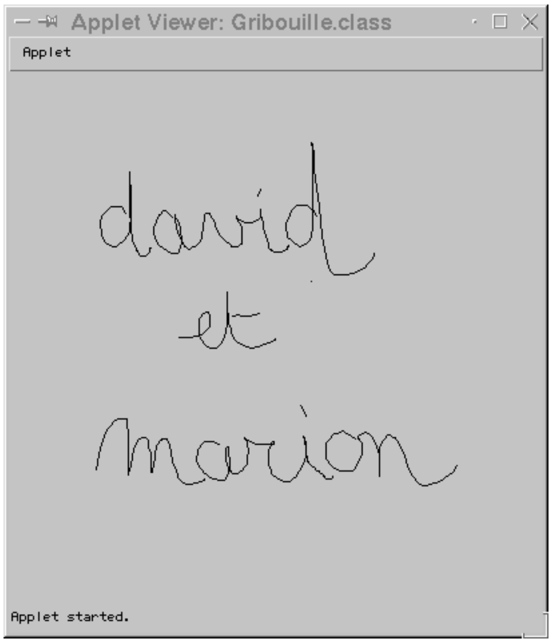
## La triplette magique paint() repaint() update()

Ce sont des méthodes de la classe `Component`.

- `paint(Graphics gc)` : ne fait rien par défaut, en pratique contient des appels de méthodes de dessin de la classe `Graphics` sur `gc`;
- `update(Graphics gc)` : par défaut efface le dessin et appelle `paint()`;
- `repaint()` : appelle `update()` en lui fournissant un contexte graphique.



## Exemple du gribouillage






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

public class Gribouille extends Applet{
  int x0,y0,x,y;

  public void init() {
    addMouseListener(new Appuyeur());
    addMouseMotionListener(new Dragueur());
  }

  public void update(Graphics gc) {
    paint(gc);
  }

  public void paint(Graphics gc) {
    gc.drawLine(x0,y0,x,y);
    x0 = x; y0 = y;
  }

  class Appuyeur implements MouseListener {
    public void mousePressed(MouseEvent e) {
      x0 = e.getX(); y0 = e.getY();
    }
    public void mouseEntered(MouseEvent e) {}
    public void mouseClicked(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
  }

  class Dragueur implements MouseMotionListener{
    public void mouseDragged(MouseEvent e) {
      x = e.getX(); y = e.getY(); repaint();
    }
    public void mouseMoved(MouseEvent e) {}
  }
}
```



- L'implémentation d'une interface demande l'écriture de *toutes* ses méthodes.
- Un *adaptateur* est une classe qui implémente une interface avec un comportement par défaut.
- Le paquetage `java.awt.event` contient un adaptateur pour toute interface auditeur qui a au moins deux méthodes.

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

public class Gribouille2 extends Applet {
  int x0,y0,x,y;

  public void init() {
    addMouseListener(new Appuyeur());
    addMouseMotionListener(new Dragueur());
  }

  public void update(Graphics gc) {
    paint(gc);
  }

  public void paint(Graphics gc) {
    gc.drawLine(x0,y0,x,y);
    x0 = x; y0 = y;
  }

  class Appuyeur extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      x0 = e.getX(); y0 = e.getY();
    }
  }
  class Dragueur extends MouseMotionAdapter {
    public void mouseDragged(MouseEvent e) {
      x = e.getX(); y = e.getY();
      repaint();
    }
  }
}
```



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

public class Gribouille3 extends Applet{
  int x0,y0,x,y;

  public void init() {

    addMouseListener(new MouseAdapter (){
      public void mousePressed(MouseEvent e) {
        x0 = e.getX(); y0 = e.getY();
      }
    });

    addMouseMotionListener(new MouseMotionAdapter (){
      public void mouseDragged(MouseEvent e) {
        x = e.getX(); y = e.getY();
        repaint();
      }
    });
  }

  public void update(Graphics gc) {
    paint(gc);
  }

  public void paint(Graphics gc) {
    gc.drawLine(x0,y0,x,y);
    x0 = x; y0 = y;
  }
}
```

Les classes `Appuyeur` et `Dragueur` sont devenues *anonymes*.



# 7

## Introduction à Swing

1. Exemples
2. La structure des JFrame
3. Menus
4. Boutons



## Premier exemple

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

class MyCloseableFrame extends JFrame{
  public MyCloseableFrame(){
    super("ma fenetre");
    setSize(300,200);
    addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent e){
        System.exit(0);
      }
    });
  }
}

public class MyCloseableFrameTest{
  public static void main(String[] args){
    JFrame frame = new MyCloseableFrame();
    frame.show();
  }
}
```

Remarques :

- On a utilisé une *classe interne anonyme* pour définir l'auditeur.
- La méthode `show()` est une méthode de la classe `Window` mère de `Frame`. Elle permet d'afficher un composant.
- Il n'est pas toujours facile de savoir si une méthode est définie dans la classe `Component`, `Window` ou `Frame`.
  Exemple de méthodes :

– Dans `java.awt.Component` :
```
boolean isEnabled()
void setEnabled(boolean)
void setVisible(boolean)
```

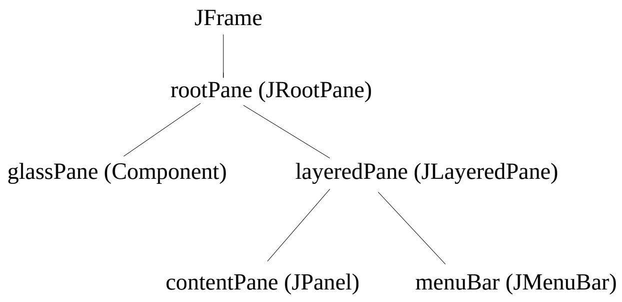
– Dans `java.awt.Window` :
```
void toFront()
void toBack()
```

– Dans `java.awt.Frame` :
```
void setTitle(String)
```

En Java, les cadres sont des conteneurs d'autres composants.

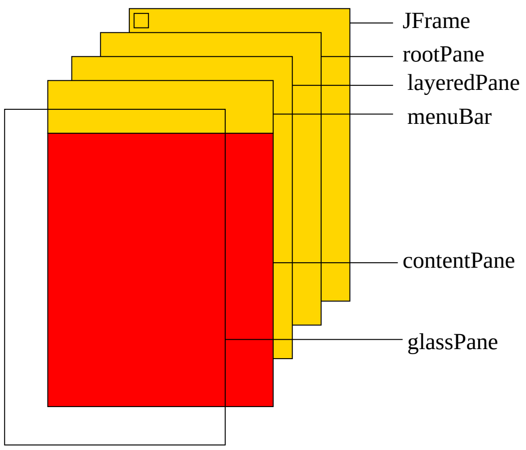


## La structure (complexe) des JFrame






## La structure (complexe) des JFrame




L'accès à la fenêtre `contentPane` se fait par la méthode `getContentPane()` de la classe `JFrame`.



## Deuxième exemple

On ne dessine pas ou on n'écrit pas directement dans une `JFrame`, on passe par son `contentPane` qui est un `JPanel`.

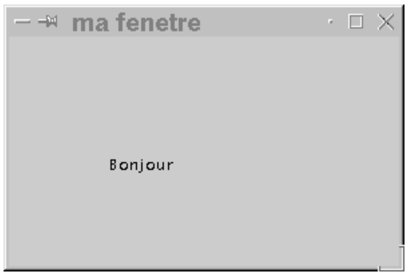
```
import java.awt.*;
import javax.swing.*;

class MyJFrame extends JFrame{
  public MyJFrame() {
    super("ma fenetre");
    setSize(300,200);
    JPanel p = new BonjourPanel();
    getContentPane().add(p);
  }
}

class BonjourPanel extends JPanel{
  public void paintComponent(Graphics gc){
    super.paintComponent(gc);
    gc.drawString("Bonjour",75,100);
  }
}

public class MyJFrameTest{
  public static void main(String[] args){
    JFrame frame = new MyJFrame();
    frame.show();
  }
}
```

- La méthode `paintComponent()` est une méthode de la classe `javax.swing.JComponent`, qui dérive de `Container`, qui dérive elle-même de `Component`.

- Chaque fois qu'une fenêtre est redessinée, le gestionnaire d'évènements Java envoie une notification à cette fenêtre. Les méthodes `paintComponent()` de tous les composants de la fenêtre sont alors exécutés.
- Il ne faut pas appeler soi-même `paintComponent()` puisque ceci est fait automatiquement.

Pour dessiner, on peut utiliser `java.awt.Graphics`.








## Actions générées par un menu

La classe abstraite `javax.swing.AbstractAction` permet de définir des actions.

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

class MyJFrameAction extends JFrame{
  public MyJFrameAction(){
    super("ma fenetre");
    setSize(300,200);
    JPanel p = new JPanel();
    getContentPane().add(p);
    Action bleuAction = new CouleurAction("Bleu",Color.blue,p);
    Action rougeAction = new CouleurAction("Rouge",Color.red,p);
    Action vertAction = new CouleurAction("Vert",Color.green,p);

    JMenu m = new JMenu("Couleur");
    m.add(bleuAction);
    m.add(rougeAction);
    m.add(vertAction);
    JMenuBar mbar = new JMenuBar();
    mbar.add(m);
    setJMenuBar(mbar);
  }
}
public class MyJFrameActionTest{
  public static void main(String[] args){
    JFrame frame = new MyJFrameAction();
    frame.show();
  }
}
```



```
class CouleurAction extends AbstractAction{
  private Component cible;
  private Color couleur;

  public CouleurAction(String nom, Color couleur, Component cible) {
    super(nom);
    this.couleur = couleur;
    this.cible = cible;
  }

  public void actionPerformed(ActionEvent e){
    cible.setBackground(couleur);
    cible.repaint();
  }
}
```

## Un exemple de JToggleButton


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

public class MyToggleFrame extends JFrame{

  public static void main(String args[]) {
    JFrame frame = new MyToggleFrame();
    frame.show();
  }

  public MyToggleFrame(){
    super("Choisir La Liaison PPP");
    getContentPane().add(new MyTogglePanel());

    Dimension dim = getToolkit().getScreenSize();
    setLocation(dim.width/2 - getWidth()/2,
                dim.height/2 - getHeight()/2);
    pack();
    setVisible(true);
```



```
    setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
    WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        int confirm = JOptionPane.showOptionDialog(
          MyToggleFrame.this,
          "Voulez-vous vraiment quitter ?",
          "Confirmation Quitter",
          JOptionPane.YES_NO_OPTION,
          JOptionPane.QUESTION_MESSAGE,
          null, null, null);
        if (confirm == 0)
          System.exit(0);
    }
    };
    this.addWindowListener(l);
  }
}


class MyTogglePanel extends JPanel {
  JPanel choixPanel, quitterPanel;

  public MyTogglePanel () {
    setLayout(new GridLayout(2,1,1,1));
    choixPanel = new JPanel(new GridLayout());
    quitterPanel = new JPanel(new GridLayout());
    add(choixPanel);
    add(quitterPanel);

    JToggleButton button1 = new JToggleButton("ppp0",true);
    Font bigFont = new Font("Dialog",Font.PLAIN, 24);
    button1.setFont(bigFont);
    choixPanel.add(button1);
    JToggleButton button2 = new JToggleButton("ppp1",false);
    button2.setFont(bigFont);
    choixPanel.add(button2);
    JToggleButton button3 = new JToggleButton("ppp2",false);
    button3.setFont(bigFont);
```



```
    choixPanel.add(button3);
    ButtonGroup buttonGroup = new ButtonGroup();
    buttonGroup.add(button1);
    buttonGroup.add(button2);
    buttonGroup.add(button3);

    ActionListener actionPPP  = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        JToggleButton b = (JToggleButton) e.getSource();
        System.out.println("liaison "+ b.getText());
      }
    };
    button1.addActionListener(actionPPP);
    button2.addActionListener(actionPPP);
    button3.addActionListener(actionPPP);


    JButton button4 = new JButton("quitter");
    button4.setFont(bigFont);
    quitterPanel.add(button4);

    ActionListener actionQuitter  = new ActionListener(){
      public void actionPerformed(ActionEvent e) {
        System.exit(0);
      }
    };
    button4.addActionListener(actionQuitter);
  }
}
```

On définit de préférence un `JPanel` qui sera ensuite inséré au choix dans une `JFrame` ou une applette.