# Cours de JAVA

Serge Rosmorduc
rosmord@iut.univ-paris8.fr

2000–2005

# Table des matières

**I Le langage JAVA. Compléments** **1**

**1 Spécificateurs** **3**
1.1 Le mot clef `final` . . . . . 3
1.2 Le mot clé static . . . . . 4
1.2.1 Champs statiques . . . . . 4
1.2.2 Méthodes statiques . . . . . 6
1.3 Accès aux champs et méthodes d'une classe . . . . . 6

**2 Héritage** **9**
2.1 Redéfinition de fonctions . . . . . 9
2.2 Implémentation de l'héritage . . . . . 9
2.3 Type et classes : le polymorphisme . . . . . 9
2.4 Classes abstraites . . . . . 10
2.5 `super` et `this` . . . . . 10
2.5.1 Introduction . . . . . 10
2.5.2 Définitions . . . . . 10
2.5.3 super et this pour qualifier des champs . . . . . 11
2.5.4 super et this dans les méthodes . . . . . 11
2.5.5 super et this dans les constructeurs . . . . . 12
2.5.6 this comme argument de méthode . . . . . 12

**3 Interfaces** **13**
3.1 Les Interfaces comme spécification . . . . . 13
3.2 Les interfaces comme collections de constantes . . . . . 13

**4 Cast** **15**
4.1 Cast et types de base . . . . . 15
4.2 Cast et classes . . . . . 15

**5 L'identité des objets** **17**
5.1 Comparaison d'objets : `equals()` . . . . . 17
5.2 Le hashcode . . . . . 18
5.3 Duplication d'un objet : `clone()` . . . . . 18

**6 Exceptions** **21**
6.1 Introduction . . . 21
6.2 Terminologie . . . 22
6.3 Les exceptions en JAVA . . . 23
6.4 Propagation d'exceptions . . . 23
6.5 Interception des exceptions en Java . . . 23
6.5.1 La clause try...catch . . . 23
6.5.2 La clause finally . . . 23
6.5.3 catch et l'héritage . . . 23
6.6 Définition d'un type d'exceptions . . . 23

**7 Threads** **25**
7.1 Introduction . . . 25
7.2 Déclaration . . . 25
7.3 Lancement d'un thread . . . 26
7.4 Le partage du temps . . . 27
7.4.1 Priorités . . . 27
7.4.2 La méthode `yield()` . . . 28
7.5 Variables partagées . . . 28
7.5.1 Synchronized . . . 29
7.5.2 Volatile . . . 30
7.6 Contrôle de l'exécution . . . 31
7.6.1 Arrêt d'un thread . . . 31
7.6.2 Attente de la fin d'un thread . . . 32
7.6.3 Méthode sleep . . . 33
7.7 Éviter les attentes actives avec `wait` et `notify` . . . 33
7.7.1 Introduction . . . 33
7.7.2 Méthodes . . . 33
7.7.3 Utilisation de wait et notify . . . 34
7.7.4 Client et serveur sans attente active . . . 35
7.7.5 Gestion d'une file de messages . . . 36
7.8 Démons . . . 38
7.9 Graphismes . . . 39
7.9.1 synchronisation et graphismes . . . 41

**II Bibliothèques particulières** **43**

**1 Les Collections** **45**
1.1 Introduction . . . 45
1.2 Itérateurs . . . 46
1.3 Organisation des classes . . . 47
1.4 L'interface Collection . . . 49

1.5 Listes . . . 50
1.6 Ensembles . . . 51
1.6.1 Ensembles triés . . . 51
1.7 Dictionnaires (`maps`) . . . 52
1.8 Piles (`Stack`) . . . 53
1.9 Exemples . . . 53
1.9.1 La collection Sac . . . 53
1.9.2 Utilisation de Map pour compter des mots . . . 54

**2 Entrées/sorties** **57**
2.1 La classe File . . . 57
2.2 Organisation des classes d'entrées/sortie . . . 60
2.2.1 Fichiers en lecture, fichiers en écriture . . . 60
2.2.2 Classes orientées octets, classes orientées caractères . . . 60
2.2.3 Filtrage . . . 60
2.3 Quelques Classes . . . 61
2.4 Exemples d'utilisation . . . 62
2.4.1 Lecture d'un fichier . . . 62
2.5 Méthodes . . . 63
2.5.1 Constructeurs . . . 63
2.5.2 Reader . . . 65
2.5.3 Writer . . . 65
2.6 La classe RandomAccessFile . . . 66
2.6.1 Ouverture et fermeture . . . 66
2.6.2 Lecture et écriture . . . 67
2.6.3 Déplacement . . . 67
2.7 Sérialisation . . . 68
2.7.1 Conditions d'emploi . . . 68
2.7.2 Champs non sauvegardés . . . 68
2.8 La classe StreamTokenizer . . . 68
2.8.1 Introduction . . . 68
2.8.2 Documentation . . . 70

**3 Java et les bases de données** **75**
3.1 Introduction à JDBC . . . 75
3.2 Architecture . . . 75
3.3 Un exemple : postgres . . . 75
3.4 établir la connexion . . . 76
3.4.1 Exemple : . . . 76
3.5 Envoyer une requête . . . 77
3.5.1 Méthodes . . . 77
3.5.2 Méthodes applicables à un **ResultSet** . . . 78
3.5.3 Execute . . . 78

3.6 Commandes préparées . . . 79
3.7 échappements SQL . . . 79
3.8 Gestion des transactions . . . 80
3.8.1 Niveau d'isolement . . . 80
3.9 Capacités de la base de données : `DataBaseMetaData` . . . 81
3.10 Exploration des tables . . . 81
3.10.1 méthodes de `ResultSetMetaData` . . . 82
3.11 Extensions du jdbc2.0 . . . 82
3.11.1 `ResultSet` navigables . . . 82

**4 Java Server Pages et Servlets 85**
4.1 Architecture d'une application web . . . 85
4.2 Introduction aux jsp . . . 85
4.2.1 principes . . . 85
4.2.2 Balises principales des jsp . . . 86
4.2.3 Les champs d'une jsp . . . 87
4.2.4 Installation des jsp . . . 87
4.3 Introduction aux servlets . . . 87
4.3.1 principe des servlets . . . 87
4.3.2 Installation des servlets . . . 87
4.3.3 Les servlets, leur vie, leur œuvre . . . 89
4.4 Les javabeans . . . 90
4.4.1 Les classes de beans . . . 90
4.4.2 Accès aux beans depuis les *jsp* . . . 91
4.4.3 accès aux beans depuis les servlets . . . 93
4.5 Architecture typique d'une application jsp . . . 94
4.5.1 Redirection vers une jsp . . . 94
4.6 Le suivi des sessions . . . 94
4.7 Création de nouvelles balises pour les jsp . . . 96
4.7.1 Une balise simple . . . 96
4.7.2 Une boucle . . . 96
4.7.3 Partage de variables . . . 96
4.8 Quelques patterns . . . 96

**5 Java et XML 97**
5.1 Le langage XML . . . 97
5.1.1 Introduction . . . 97
5.1.2 Documents XML . . . 97
5.1.3 XSLT . . . 101
5.1.4 Schémas XML . . . 103
5.2 Les API Java . . . 103
5.2.1 Document object model (DOM) . . . 103
5.2.2 Simple API for Xml Parsing (SAX) . . . 107

5.3 XSL et java . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
5.4 Quelques outils java utilisant XML . . . . . . . . . . . . . . . . . . . . . . . . 111
5.4.1 SVG et Batik . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111
5.5 Appendice : un fragment de la description du langage XML . . . . . . . . . . . . 111

**6 Swing Avancé 113**
6.1 Patterns en swing : l'exemple de `JTable` . . . . . . . . . . . . . . . . . . . . . 113
6.1.1 Le pattern Observateur . . . . . . . . . . . . . . . . . . . . . . . . . 113
6.2 Le pattern *commande* en swing : actions et edits . . . . . . . . . . . . . . . . . 115
6.2.1 Actions et le pattern *commande* . . . . . . . . . . . . . . . . . . . . . 115
6.2.2 Undo, classes textes et design pattern commande . . . . . . . . . . . . . 115

# Première partie

# Le langage JAVA. Compléments

# Chapitre 1

# Spécificateurs

Un certain nombre de mots clefs permettent de spécifier le comportement d'un champ ou d'une méthode. Nous détaillerons ici `final, static`, ainsi que les opérateurs de visibilité `public, private, protected`.

## 1.1 Le mot clef **final**

Le mot clef final se place dans la déclaration d'un champ, d'une classe ou d'une méthode.Il indique que l'élément déclaré ne peut être modifié.

Dans le cas d'un champ, il permet donc de définir des constantes. On l'utilise généralement avec `static` (voir 1.2). Exemple :

```
class Cercle {
  private double x,y,r;
  static final double PI= 3.14;
  ...
  public double getPerimetre() {
    return 2*PI*r;
  }
}
```

Il est aussi possible de l'utiliser si la valeur d'un champ est fixe pour un objet donné. Par exemple, un objet de classe `String` n'est pas modifiable. Une définition possible de cette classe serait donc :

```
public class MyString {
    private final char rep[];
    public MyString(char t[])
    {
        // Initialisation de rep
        rep= new char[t.length];
        for (int i= 0; i< t.length; i++)
            rep[i]= t[i];
    }
```

```
   ...
}
```

Notez que le final de l'exemple précédent n'interdit pas de changer la valeur d'une des cases du tableau rep. Il interdit uniquement faire pointer `rep` sur un autre tableau. Ainsi, le code suivant ne compilera pas :

```
public class MyString {
    ...

    public void set(char t[]) {
        rep= t;
        // Erreur à la compilation :
        // "Can't assign a second value to a blank final variable: rep"
    }
}
```

Dans le cas d'une méthode, `final` interdit la redéfinition de la méthode par héritage.

Dans le cas d'une classe il interdit d'en dériver des sous-classes.

## 1.2 Le mot clé static

Le mot-clé `static` vient du C. Il caractérise dans ce langage des variables dont la caractéristique la plus marquante est de n'exister *qu'en un seul exemplaire*.

En JAVA, le mot-clé `static` indique qu'un élément d'une classe est lié à la *classe* et non à un *objet* particulier.

### 1.2.1 Champs statiques

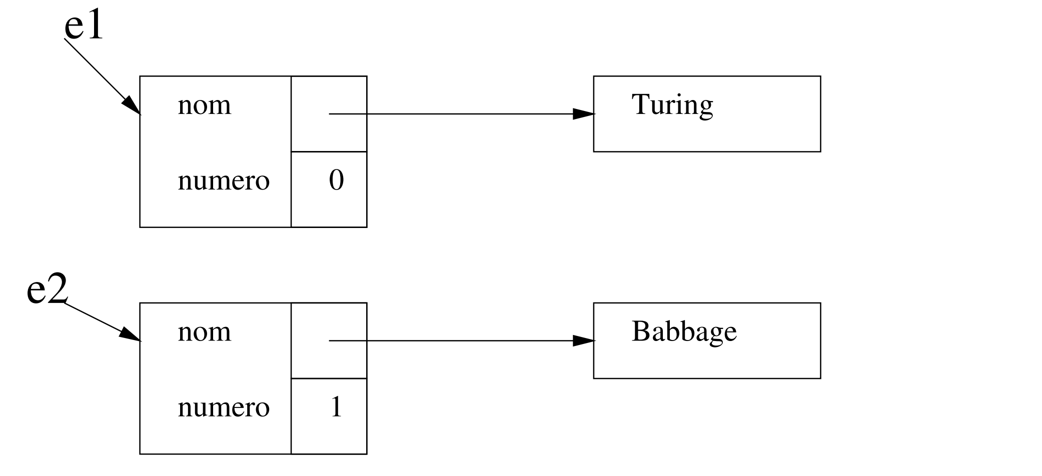
Normalement, un champ dans une classe est lié à un objet. Par exemple, soit la classe `Etudiant`:

```
public class Etudiant {
  String nom;
  int numéro;
  ...
  public Etudiant(String n) {nom= n;}
  public void setNumero(int n) {numero= n;}
}
```

Le `numéro` d'un objet étudiant est lié à cet objet. Si nous considérons deux objets étudiants différents, `e1` et `e2`, leurs numéros seront deux données *distinctes* :

```
  e1= new Etudiant("Turing"); e1.setNumero(0);
  e2= new Etudiant("Babbage"); e2.setNumero(1);
```

**FIG. 1.1 –** Champs dans des objets d'une même classe




l'espace mémoire correspondant à `e1` et celui correspondant à `e2` contiennent chacun une place pour le numéro et une place pour le nom, comme le montre la figure 1.1.

Dans le cas de champs statiques, l'espace mémoire pour le champ est réservé *indépendamment des objets* (figure 1.2). Il existe donc même si aucun objet de la classe n'existe ; de plus il existe en un seul exemplaire. On y accède en écrivant

```
NomDeLaCLASSE.nomDuChamp
```

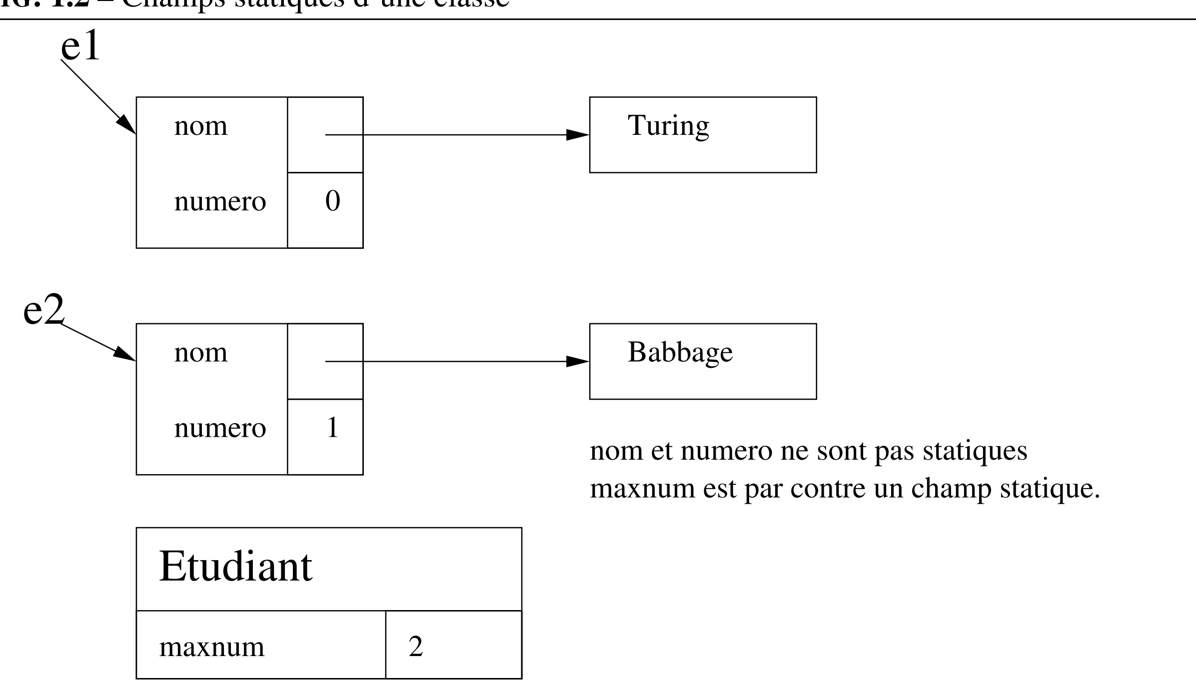
**Utilisation d'un champ statique :** Un champ statique sert généralement pour conserver des caractéristiques de la classe toute entière. Il s'agit souvent de constantes (champs `final static`). Il peut cependant s'agir de véritables variables. Par exemple, supposons que nous écrivions d'une classe Etudiant (au hasard) et que nous voulions que le numéro d'étudiant soit automatiquement incrémenté à chaque nouvel étudiant créé. Nous pourrions écrire :

```
public class Etudiant {
  static int maxnum= 0;
  int numero;
  String nom;
  ...

  public Etudiant(String nom) {
    numero= maxnum;
    maxnum= maxnum+1;
    this.nom= nom;
  }
}
```

Remarquons cependant qu'il serait bien meilleur de créer une classe pour gérer les numéros d'étudiants.

**FIG. 1.2 –** Champs statiques d'une classe




### 1.2.2 Méthodes statiques

Une méthode statique est elle aussi indépendante des objets. Elle fonctionne donc comme les fonctions des langages non orientés objets comme C. On les appelle en écrivant :

```
NomDeLaCLASSE.nomDeLaFonction(Arguments)
```

Dans les bibliothèques mathématiques de Java, les fonctions usuelles sont déclarées statiques ; en effet, elles ne s'appliquent pas à un objet :

```
x= Math.cos(1.4);
```

Lorsqu'un programme Java est lancé, il n'existe *a priori* aucun objet, donc aucun élément auquel appliquer une méthode. C'est pour cela que les concepteurs du langage ont décidé que le corps d'un programme serait constitué par une méthode statique : `main`.

## 1.3 Accès aux champs et méthodes d'une classe

**public** le champ est accessible de partout ; en pratique, réservez ce statut aux méthodes. La seule exception raisonnable pour les champs, ce sont les constantes.

**private** le champ n'est accessible que depuis la classe elle-même. Si vous n'avez pas une excellente raison de donner un autre statut à un champ ou à une méthode, choisissez `private`.

**(rien)** le champ est accessible depuis toutes les classes du *package* ; ça n'est pas une raison pour en abuser :-). En pratique, il arrive parfois que plusieurs classes soient intimement liées. Typiquement, elles n'ont pas de sens les unes sans les autres. Dans ce cas, ce type d'accessibilité est envisageable (voir par exemple 1.9.1 pour un usage possible).

**protected** un champ `protected` d'une classe `A` n'est accessible que depuis le *package* et les classes dérivées de `A`. Typiquement, la classe `A` définit une fonctions ou des champs qui seront utiles aux classes dérivées, et seulement à celles-ci.

# Chapitre 2

# Héritage

L'*héritage* est un mécanisme qui permet d'étendre une classe en lui *ajoutant* des possibilités ou en *redéfinissant* certaines des fonctionnalités existantes.

Concrètement, étant donné une classe `A`, l'héritage permet de créer une classe `B`, qui a toutes les caractéristiques de la classe `A`, plus éventuellement de nouveaux champs et de nouvelles méthodes.

## 2.1 Redéfinition de fonctions

## 2.2 Implémentation de l'héritage

## 2.3 Type et classes : le polymorphisme

Introduisons quelques définitions [1] :

**type** un objet a un type et un seul : celui qui lui est donné par son constructeur.

**classe** un objet peut par contre avoir *plusieurs* classes : celle correspondant à son type, ainsi que toutes les sur-classes de celle-ci.

Supposons que la classe `Etudiant` hérite de `Personne` qui hérite de `Object`. Alors :

```
Personne e= new Etudiant("toto");
```

crée un objet de type Etudiant, qui appartient aux classes `Etudiant`, `Personne` et `Object`. Par contre `e` est de classe `Personne`, ce qui est légal car c'est bien une des classes de l'objet.

**redéfinition (overriding)** c'est le fait de réécrire dans une classe B une méthode d'une des classes dont B hérite. Lorsque cette méthode sera appelée sur un objet de *type* B, alors c'est la méthode redéfinie qui sera utilisée et non la méthode de la classe ancêtre.

1. L'usage des mots *type* et *classe* dans ce sens n'est à notre connaissance pas standard. Cette terminologie provient du livre *Design Patterns*.

**surcharge (overloading)** à ne pas confondre avec le précédent. c'est le fait d'utiliser le même nom pour des méthodes différentes. Quand les méthodes sont dans la même classe, elles sont distinguées par leurs arguments. Notez que la surcharge *n'a aucun rapport* avec l'héritage [2].

**polymorphisme**

## 2.4 Classes abstraites

## 2.5 `super` et `this`

### 2.5.1 Introduction

En Java, il arrive parfois qu'une variable en « cache » une autre. Très classiquement, considérons une classe `Etudiant` et un de ses constructeurs possibles :

```
public class Etudiant {
  String nom;
  public Etudiant(String nom) {
    ???
  }
}
```

Dans le corps du constructeur, le paramètre « `nom` » cache le champ (ou variable d'instance) « `nom` ». Dans 99 % des langages de programmation, le problème est résolu en changeant le nom du paramètre. C'est là qu'intervient `this`. « `this` » signifie « l'objet auquel s'applique la méthode », ou, dans le cas d'un constructeur, « l'objet en train d'être construit ». Dans ce cas, l'écriture « `this.nom` » n'est pas ambiguë, et désigne bien « le champ `nom` de l'objet de classe `Etudiant` que nous construisons ».

Cette fonction, pour être très utilisée en Java, n'en est pas moins secondaire : on pourrait fort bien s'en passer. Il est par contre des fois où l'emploi de `this` ou de son compère `super` est primordial.

### 2.5.2 Définitions

**this** signifie « l'objet auquel s'applique la méthode », ou, dans le cas d'un constructeur, « l'objet en train d'être construit » ;

**super** S'emploie dans le cas d'une classe *`B`* qui étend une classe de base *`A`*. Il signifie « l'objet auquel s'applique la méthode », ou, dans le cas d'un constructeur, « l'objet en train d'être construit », mais *vu comme un objet de classe `A`.*

2. La surcharge en java est un concept simple. En C++, nous verrons qu'elle est bien plus complexe, mais aussi plus puissante

### 2.5.3 super et this pour qualifier des champs

**this :** quand un champ d'un objet est caché par une variable locale ou un argument, le mot clef `this` permet de lever l'ambiguïté et de désigner le champ. Classiquement :

```
public class Etudiant {
  String nom;
  public Etudiant(String nom) {
    this.nom= nom;
    // this.nom : champ nom de this
    // nom : argument du constructeur.
  }
}
```

**super :** si une classe fille définit un champ dont un homonyme existe dans une classe parente, ce dernier est masqué. Soit par exemple le code :

```
class A {
  public int a;
}

class B extends A {
  public String a;
  ...
}
```

le champ `a` de type `String` de `B` masque le champ `int` de `A`. Le mot-clef `super` permet de faire référence, dans une méthode de la classe `B,` à un objet de classe `B` en le considérant comme un objet de la classe parente, c'est-à-dire `A`. Ainsi :

```
class B extends A {
  String a;
  ...
  public void afficher() {
    System.out.println(a); // affiche le champ String a
    System.out.println(super.a); // affiche le champ int a
                                 // hérité de A.
  }
}
```

Notez que cet utilisation de `super` est très rare. En effet, il est formellement déconseillé d'accéder directement aux champs d'une classe. Normalement, on aurait dû écrire des accesseurs.

### 2.5.4 super et this dans les méthodes

**this :** il est licite d'écrire

```
this.appelDeMethode();
```

mais comme c'est équivalent à

```
appelDeMethode();
```

c'est parfaitement inutile.

**super :** l'idée est toujours la même. Supposons que nous ayons :

```
class Personne {
  ...
  public void afficher() {
    // affiche les données relative à une personne :
    // son nom, son prénom, son adresse.
    ...
  }
}

class Etudiant extends Personne {
  ...
  public void afficher() {
    // affiche les données relative à un étudiant :
    // d'abord les données existantes dans la classe Personne,
    // puis celles spécifiques aux étudiants.
  }
}
```

Il serait intéressant de pouvoir utiliser la méthode `afficher` de `Personne` pour écrire celle de `Etudiant`.

### 2.5.5 super et this dans les constructeurs

### 2.5.6 this comme argument de méthode

# Chapitre 3

# Interfaces

## 3.1 Les Interfaces comme spécification

Le mécanisme de l'héritage en Java est limité à l'héritage simple. Une classe a une parente directe et une seule (sauf `Object,` qui n'en a pas du tout). L'héritage a un sens très fort : « B hérite de A » signifie « tout B est un A ». Or, il arrive que l'on veuille simplement signifier qu'une classe « sait faire » telle ou telle chose. C'est le sens des interfaces.

Une interface est principalement un ensemble d'en-têtes de méthodes. Une classe peut *implémenter* une ou plusieurs interfaces, c'est à dire fournir le code correspondant aux méthodes de l'interface.

## 3.2 Les interfaces comme collections de constantes

Les interfaces peuvent aussi contenir des constantes. Il suffit alors d'implémenter l'interface pour pouvoir les utiliser. Considérons par exemple l'interface suivante :

```
  public interface CodesJours {
    int DIMANCHE= 0;
    int LUNDI= 1;
    int MARDI= 2;
    int MERCREDI= 3;
    int JEUDI= 4;
    int VENDREDI= 5;
    int SAMEDI= 6;
}
```

Notez que ces champs sont en réalité `static` et `final`, mais qu'il est optionnel de l'écrire.

Toute classe qui implémente `CodesJours` dispose directement des constantes qu'elle définit :

```
class Calendrier implements CodesJours {
    ...
    initWeekEnd() {
```

```
        // Notez que SAMEDI est utilisé directement
        jour[SAMEDI].setRepos(true);
        jour[DIMANCHE].setRepos(true);
    }
    ...
}
```

# Chapitre 4

# Cast

L'opération de conversion (casting) couvre en Java deux concepts différents. Elle se comporte en effet différemment sur les types de base, où il s'agit d'une véritable conversion, et sur les classes, où c'est plutôt une opération syntaxique.

## 4.1 Cast et types de base

Les divers types de base (int, double, etc...) sont plus ou moins compatibles entre eux. On conçoit par exemple qu'il soit logique de pouvoir écrire un entier dans un double. Java permet d'ailleurs cette opération sans autre forme de procès :

```
int i= 1;
double y= i;
```

Par contre, certaines conversions causent une perte d'information. Écrire un `double` dans un `int,` c'est en perdre la partie décimale. De même, écrire un `int` dans un `short` (entier court) peut conduire à une perte de données.

En Java, les conversions avec perte de donnée potentielle sont possibles, mais, contrairement aux conversions sans perte de données, elles doivent être explicitement marquée par l'opération de `casting.`

Un cast permet de convertir une donnée d'un type de base en un autre. Il s'écrit en faisant précéder la valeur à convertir par le nom du type cible de la conversion, écrit entre parenthèses :

```
double y= 1.3;
int j= (double)y;
```

## 4.2 Cast et classes

Pour reprendre le vocabulaire de 2.3,

# Chapitre 5

# L'identité des objets

Le problème de l'identité est simple à première vue : il s'agit de savoir si deux objets sont « les mêmes ». Néanmoins, les apparences sont trompeuses. Par défaut pour java chaque objet créé est différent des autres.

## 5.1 Comparaison d'objets : `equals()`

La méthode `equals` est héritée par toutes les classes. La classe `Objet` définit `equals` comme :

```
public boolean equals(Objet o)
{
  return this == o;
}
```

C'est donc initialement un équivalent de ==. Pour les classes qui veulent disposer d'une définition différente de l'égalité, on doit redéfinir `equals`. C'est ce que fait, par exemple, la classe `String`.

Exemple de redéfinition :

Encapsulation d'un entier

```
class Entier {
    int val;

    public Entier(int v) {val= v;}

    public boolean equals(Object o) {
        // On compare un Entier et un Object.
        // Ils peuvent être de type différent :
        if (o instanceof Entier)
        {
            // o est un Entier. Le récupérer comme tel
            // grâce à un cast :
            Entier e= (Entier)o;
            return (e.val == val);
```

```
        }
        else
          // o n'est pas un Entier. Donc il ne peut être égal à this
          return false;
    }

    // Fonction hashCode. Voir ci-dessous.
    public int hashCode() {
        return val;
    }

    public int getValeur() {return val;}
    public void setValeur(int v) {val= v;}
}
```

## 5.2 Le hashcode

Si une classe redéfinit `equals`, elle *doit* aussi redéfinir la méthode `hashcode` :

`public int` **`hashcode`** `()`
: fonction renvoyant un entier, utilisée par les classes comme `Hashtable`, `HashMap` et `HashSet`.

  La contrainte est que si deux objets sont égaux au sens de `equals`, ils doivent avoir le même hashcode (la réciproque n'étant pas forcément vérifiée : deux objets peuvent avoir le même code, mais être différents). Par contre, plus le `hashcode` différencie des objets différents, plus il est efficace.

Notez qu'une fonction constante remplit les conditions (mais ne fournit pas un « bon » `hashcode`.

Si `hashcode` et `equals` sont en désaccord, les classes telles que `HashTable`, qui utilisent le `hashcode`, donneront des résultats erronés.

## 5.3 Duplication d'un objet : `clone()`

à rédiger

```
public class Etudiant implements Cloneable
{
    private String nom;

    protected Object Clone() {
        try {
            Etudiant e = (Etudiant)super.clone();
            e.nom = (String)nom.clone();
            return e;
```

```
        } catch (CloneNotSupportedException e) {
            // Impossible
        }
    }
}
```

**Important** `clone()` ne peut ni utiliser un constructeur de la classe à cloner, ni appeler `new` pour construire son résultat.

# Chapitre 6

# Exceptions

**BON EXEMPLE POUR LES EXCEPTIONS : la classe Note.**

## 6.1 Introduction

Dans de nombreux langages de programmations, comme C, la gestion d'erreurs est un domaine délicat et, disons-le, rebutant. C'est sans doute la raison pour laquelle de nombreux programmeurs la négligent, ce qui peut aboutir à des problèmes graves. Ainsi, par exemple, la plupart des failles de sécurités dans les systèmes UNIX sont causées par l'exploitation d'une gestion d'erreur défectueuse.

Pourquoi la gestion d'erreur pose-t-elle des problèmes ? Considérons l'exemple suivant : soit un programme qui ouvre un fichier, y lit trois entiers, puis le ferme.

L'algorithme est donc le suivant :

```
f= ouvrir_fichier(nom);
i1= lire_entier(f);
i2= lire_entier(f);
i3= lire_entier(f);
fermer f;
```

La lecture de ce code ne pose aucun problème, et l'on voit tout de suite ce qu'il fait. Malheureusement, à chaque étape, un problème peut se poser : le fichier peut ne pas exister ou ne pas être consultable. D'autre part, il peut contenir des données erronées, et la lecture de i1, i2 et i3 peut échouer. Regardons à quoi correspondrait un système gérant les erreurs :

```
si ((f= ouvrir_fichier(nom)) == 0)
   affiche "fichier non ouvert"
   exit 1
i1= lire_entier(f);
si erreur_de_lecture
   affiche "fichier non ouvert"
   exit 1
i2= lire_entier(f);
si erreur_de_lecture
```

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com



```
     affiche "fichier non ouvert"
     exit 1
  i3= lire_entier(f);
  si erreur_de_lecture
     affiche "fichier non ouvert"
     exit 1
  fermer f;
```

On s'aperçoit que ce code est plus lourd, et bien moins lisible que le code précédent.

Le mécanisme des exceptions permet de *séparer* le code qui décrit le déroulement « normal » du programme et le code de gestion d'erreur. C'est ce que fait l'instruction `try ... catch` :

```
try
  code
  décrivant
  le déroulement normal du programme
catch type d'erreur 1 traitement de l'erreur 1
catch type d'erreur 2 traitement de l'erreur 2
```

Ainsi, notre lecture de fichier s'écrirait :

```
try
  f= ouvrir_fichier(nom);
  i1= lire_entier(f);
  i2= lire_entier(f);
  i3= lire_entier(f);
  fermer f;
catch fichier non ouvert afficher fichier non ouvert
catch entier non lu fermer f; afficher fichier mal formé
```

Le principe est le suivant : `ouvrir_fichier` ou `lire_entier`, si elles échouent, *lèvent une exception*, c'est à dire envoient un message d'erreur. Cela interrompt le programme en cours. Par exemple, si la lecture de `i1` échoue, les lignes suivantes ne seront pas exécutées. Au lieu de cela, le contrôle passe aux blocs `catch` qui suivent le `try`, et le code correspondant à l'exception levée (dans notre exemple, `entier non lu`) est exécuté.

**Propagation :** rien n'oblige, de plus, à gérer une erreur dans la fonction même où elle se produit. Si une exception n'est pas traitée dans la méthode où elle est levée, elle se *propage,* en remontant le fil des appels de méthodes. En fin de compte, si elle n'est jamais traitée, elle interrompt l'exécution du programme.

## 6.2 Terminologie

**Exception :** objet représentant un message d'erreur ;

**Levée d'une exception :** envoi d'un message d'erreur, qui interrompt le cours normal du programme ;

**Propagation d'une exception :** mécanisme par lequel une exception est transmise de fonction en fonction ;

**Traitement d'une exception :** interruption de l'exception, permettant éventuellement de reprendre le cours du programme.

## 6.3 Les exceptions en JAVA

Certaines méthodes peuvent déclencher des exceptions. La documentation de l'API java l'indique alors :

```
public static int parseInt(String s)
        throws NumberFormatException
```

un appel de parseInt peut lever une exception si la valeur de s n'est pas correcte

Le programmeur qui appelle une telle méthode doit préciser s'il *intercepte* l'exception ou la laisse se *propager*.

## 6.4 Propagation d'exceptions

Quand un programmeur appelle une méthode qui *peut* déclencher une exception, et qu'il décide de ne pas intercepter celle-ci, il *doit* le préciser. Si, dans une méthode, mettons `lireFichier()`, on appelle la méthode `parseInt` évoquée précédemment, il est

## 6.5 Interception des exceptions en Java

### 6.5.1 La clause try...catch

### 6.5.2 La clause finally

### 6.5.3 catch et l'héritage

## 6.6 Définition d'un type d'exceptions

* exceptions :

try... catch, throws, throw

- organisation des exceptions

By convention, class Throwable and its subclasses have two constructors, one that takes no arguments and one that takes a String argument that can be used to produce an error message.

try int a[] = new int[2]; a[4]; catch (ArrayIndexOutOfBoundsException e) System.out.println("exception: " + e.getMessage()); e.printStackTrace();

Error : non rattrapables

Exception : The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.

RuntimeException : Ces exceptions n'ont pas forcément besoin d'être rattrapées.

Error LinkageError NoClassDefFoundError Exception IOException EOFException FileNotFoundException RuntimeException ArithmeticException NumberFormatException IndexOutOfBoundsException ArrayIndexOutOfBoundsException StringIndexOutOfBoundsException NullPointerException

- liste de catch - finally

# Chapitre 7

# Threads

## 7.1 Introduction

**Définition :** Si plusieurs parties d'un programme s'exécutent en même temps (ou semblent s'exécuter en même temps), on dit qu'elles s'exécutent en *parallèle*. Chaque exécution de partie de programme est appelée un *thread* (ou *fil* d'exécution).

Les `threads` permettent donc d'écrire des programmes multitâches. Par défaut, un programme *Java* est composé d'un seul `Thread.`

À la différence des processus d'Unix, les threads d'un même programme partagent le même espace mémoire, et communiquent facilement entre eux.

## 7.2 Déclaration

Il existe deux méthodes pour exécuter des threads. Une première méthode est d'étendre la classe `Thread.` Dans ce cas, il faut redéfinir la méthode run. Au lancement du *thread,* c'est cette méthode qui sera exécutée.

```
// Thread héritant de la classe thread :

public class  T1 extends Thread {
    volatile private T2 k;

    public void setK(T2 k) {
        this.k= k;
    }

    // Ce thread affiche de manière répétée la valeur de
    // la variable k

    public void run() {
        while (true) {
            System.out.println(k);
            try {
```

```
                // Attend une milliseconde
                sleep(1);
            } catch (InterruptedException e) {}
        }
    }
}
```

La seconde méthode est d'implémenter l'interface `Runnable`. Comme précédemment, il s'agit en fait d'écrire une méthode `run()`

```
public class T2 implements Runnable {
    private int nb= 0;

    // Compte jusquà 1000000
    public void run() {
        while (nb < 100000000) {
                nb++;
            }
    }

    public String toString() {
        return "" + nb;
    }
}
```

## 7.3 Lancement d'un thread

Un thread se lance grâce à la méthode `start()`. Une fois le thread lancé, sa méthode `run()` s'exécute. Lorsque cette méthode se termine, le thread meurt.

Un thread défini par héritage (première méthode) se lance de la manière suivante :

```
// Création d'un objet thread :
MonThread t1= new MonThread();
// Pour l'instant, rien ne se passe.
// On lance le thread :
t1.start();
```

Pour lancer un thread défini par implémentation (seconde méthode), il faut déclarer deux objets : un objet `o1` de la classe implémentant `Runnable`, et l'autre de classe `Thread`. Ce dernier prendra `o1` comme paramètre de son constructeur. Il suffit ensuite d'appeler la méthode `start()` du thread.

```
// On crée un objet de la classe implémentant Runnable :
MonRunnable monrun= new MonRunnable();
// On crée un thread
Thread secondThread= new Thread(monrun);
// On lance le thread
secondThread.run();
```

Une fois les threads lancés, leur exécution se fait en parallèle. Notez que le résultat d'une telle exécution n'est pas déterministe, et dépend de la machine virtuelle java.

Voici un exemple utilisant les deux classes définies précédemment :

```
public class Th1 {
    public static void main(String arg[]) {
        T1 t1= new T1();
        T2 t2= new T2();
        Thread t3= new Thread(t2);
        t1.setK(t2);
        t1.start(); t3.start();
        while (true) {
            System.out.println("ici main");
            try {
                Thread.currentThread().sleep(10000);
            } catch (InterruptedException e) {}
        }
    }
}
```

Une exécution de cet exemple affiche :

```
[rosmord@djedefhor Cours]% java Th1
ici main
0
17489761
ici main
35044086
ici main
100000000
100000000
100000000
100000000
etc...
```

## 7.4 Le partage du temps

Sur la plupart des systèmes, le multitâche est *simulé.* Sur un système monoprocesseur, à un instant $t$ donné, un seul thread est en train de tourner. Pour donner l'impression du multitâche, un système comme comme Unix interrompt assez souvent le processus « qui tourne », puis choisit un nouveau processus, qui tourne à son tour pendant quelques millisecondes, et ainsi de suite.

On appelle *ordonnanceur* (en anglais : *scheduler*) le programme qui choisit le processus qui va tourner.

### 7.4.1 Priorités

Les priorités aident l'ordonnanceur de la machine virtuelle java à choisir le thread qui va tourner. Parmis tous les threads candidats, java choisit l'un de ceux qui ont la priorité la plus

importante. La priorité est un entier entre 0 et 10, associé à un thread. On peut la consulter et la modifier grâce à deux méthodes de `Thread`, `getPriority` et `setPriority`.

`int` **`getPriority`** `()`
Récupère la priorité actuelle de l'objet thread auquel la méthode est appliquée.

`void` **`setPriority`** `(int pri)`
`throws` **`SecurityException, IllegalArgumentException`**
fixe la priorité de l'objet thread auquel la méthode est appliquée. la valeur de `pri` doit être comprise entre `Thread.MIN_PRIORITY` (0) et `Thread.MAX_PRIORITY` (10).

### 7.4.2 La méthode `yield()`

Java ne garantit absolument pas que deux threads seront vraiment exécutés en parallèle. En général, sur une machine monoprocesseur, à un instant donné, seul un des threads tourne. Le multitâche est *simulé* en interrompant fréquemment le thread qui tourne pour passer la main à l'autre. Un observateur externe a l'impression que les deux threads fonctionnent en même temps.

Le problème est double :

1. interrompre le thread qui tourne ;
2. choisir un thread qui va tourner

Les spécifications de java ne précisent pas quand cela doit se produire. Il est donc en théorie possible que dans certaines implémentations un thread ne rende jamais la main et que seul un des threads puisse donc tourner. Pour éviter cela, java propose la méthode `yield()`. Le thread sur lequel on appelle cette méthode rend la main. La machine virtuelle choisit alors un thread à exécuter parmis les threads exécutables de plus haute priorité. Il faut noter qu'à certain moments, un thread, même de haute priorité, peut ne pas être exécutable (par exemple, s'il attend un événement) ; il est donc possible qu'un thread tourne alors qu'il existe des threads de plus haute priorité, si ceux-ci sont bloqués.

## 7.5 Variables partagées

Comme nous l'avons dit en introduction, la grande différence entre les threads et les processi Unix réside dans la gestion de la mémoire. La mémoire d'un processus Unix est *a priori* inaccessible depuis les autres processi. Grâce à cela, un programme Unix mal écrit ne peut pas « polluer » l'espace mémoire des autres programmes. De plus, à la fin d'un programme Unix, toute la mémoire utilisée est liberée. Par contre, comme les espaces mémoire sont séparés, il n'est pas possible de les utiliser pour communiquer entre processi [1]

1. Il est cependant possible d'utiliser de la mémoire partagée, mais il faut le demander explicitement.

L'espace mémoire utilisé par les threads est, quand à lui, partagé. Tous les threads d'un même programme peuvent potentiellement modifier et lire toutes les variables utilisée dans le programme. Cela permet à plusieurs threads de communiquer entre eux par le biais des variables, en modifiant ou en lisant leurs valeurs (voir le dernier exemple de 7.3, dans lequel les deux threads t1 et t3 lisent et modifient la variable `t2.nb`).

L'ennui, c'est que l'accès en parallèle à une même ressource pose des problème théoriques. Considérons les deux pseudo-codes suivants :

```
thread 1
1  a= x;
2  a++;
3  x= a;
```

```
thread 2
1  b= x;
2  b--;
3  x= b;
```

Supposons que ces deux threads s'exécutent en parallèle, et que x soit une variable partagée entre les deux. Supposons de plus que la valeur initiale de `x` soit 0.

Si le thread 1 s'exécute d'abord, et le thread 2 ensuite, `x` vaudra 0 au final. Supposons par contre le comportement suivant (1.3 désigne la ligne 3 du thread 1) :

```
1.1   a= x;
2.1   b= x;
1.2   a++;
2.1   b--;
1.3   x= a;
2.3   x= b;
```

Alors, x vaudra -1. En supposant que, par contre, 1.3 s'exécute *après* 2.3, `x` vaudra +1.

Bref, selon le comportement du programme, qui est absolument imprévisible, `x` vaudra soit 0, soit 1, soit -1. Ce comportement aléatoire a peu de chance de correspondre à ce que recherchait le programmeur.

La solution utilisée est de garantir que certaines portions de code ne seront pas exécutées en parallèle. On considère la variable `x` comme une ressource. Lors d'une écriture, *un seul* processus doit y avoir accès.

## 7.5.1 Synchronized

La déclaration `synchronised` permet de *verrouiller* un objet : si deux fonctions sont synchronisées sur un objet, elles ne pourront s'exécuter en parallèle. Il existe deux possibilités d'utilisation de `synchronized` : l'une dans le corps du code, l'autre dans une déclaration de méthode.

### Code synchronized

La déclaration `synchronized` permet de *verrouiller* un objet. Elle a la forme :

```
synchronized (obj) {
  code
  .....
}
```

Son effet est le suivant : pour exécuter le code, la méthode doit verrouiller l'objet. Si celui-ci n'est pas déjà verrouillé, tout va bien. Sinon, l'exécution du thread est *suspendue* jusqu'à ce que l'objet soit libéré par l'autre thread qui l'utilisait en le verrouillant. À ce moment, un des threads qui veulent verrouiller l'objet obtient le verrou, et les autres threads qui veulent un verrou restent suspendus.

Pour rendre déterministe le code précédent, il suffirait donc d'écrire :

thread 1

```
1  synchronized (x) {
2    a= x.getValue();
3    a++;
4    x.setValue(a);
5  }
```

thread 2

```
1  synchronized (x) {
2    b= x.getValue();
3    b--;
4    x.setValue(b);
5  }
```

### Méthode synchronized

On peut aussi synchroniser le code d'une méthode toute entière. Dans ce cas, l'objet verrouillé est celui auquel la méthode s'applique. Cela s'écrit :

```
public void synchronized increment() {
  int a= getValue();
  a++;
  setValue(a);
}
```

### Fonctionnement de synchronized

À chaque objet *Java* est associé un verrou. Une méthode ou un bout de code `synchronized` doit *obtenir* ce verroux pour tourner. Une fois que le verroux est obtenu, il est conservé jusqu'à la fin de la méthode ou de la section `synchronized.`

Si un thread essaie d'exécuter du code `synchronized` alors que le verroux nécessaire n'est pas disponible, le thread est mis en attente. Lorsque le verroux sera disponible, il pourra peut-être l'obtenir.

En pratique, dans la plupart des cas, si un objet est utilisé par plusieurs threads à la fois, il vaut mieux que les méthodes de cet objet soient `synchronized.` Si cela n'est pas le cas, on peut utiliser des blocs `synchronized` ou alors encapsuler la classe réelle de l'objet dans une classe dont les méthodes seront synchronisées.

## 7.5.2 Volatile

Le compilateur peut conserver une copie locale d'une variable, pour des raisons d'efficacité. Normalement, on n'a pas de problème si on verrouille les variables correctement.

Quand une variable est volatile, toute modification ou consultation est effectivement réalisée sur la variable partagée.

```
class Test {
```

```
  static volatile int i = 0, j = 0;
  static void one() { i++; j++; }
  static void two() { System.out.println("i=" + i + " j=" + j); } }
}
```

Si one() et two() sont exécutées concurremment, sans « volatile », two() peut éventuellement observer des cas où j serait plus grand que i. (par exemple, si la valeur de "j" est récupérée, mais pas celle de "i"). Avec volatile, non.

Cependant, la seule méthode vraiment sûre est de passer par `synchronized` !

## 7.6 Contrôle de l'exécution

### 7.6.1 Arrêt d'un thread

Pour arrêter un thread, la marche à suivre est celle de l'exemple suivant :

arrêt de thread

```
// démo d'arrêt des threads

// Thread héritant de la classe thread :
class  T4 extends Thread {
    boolean continuer= true;

    public synchronized void arreter() {
        continuer= false;
    }

    public synchronized boolean getContinuer() {
        return continuer;
    }

    public void run() {
        int k=0;
        while (getContinuer()) {
            System.out.println("hello");
            yield();
        }
    }
}

public class Th3 {
    public static void main(String arg[]) {
        T4 t= new T4();
        t.start();
        while (true) {
            System.out.println("ici main");
            try {
                Thread.currentThread().sleep(1000);
            } catch (InterruptedException e) {}
```

```
            t.arreter();
        }
    }
}
```

Le thread t tourne tant que sa variable interne continuer est true. En la mettant à false, on arrête t. Notons cependant que cet arrêt n'est pas brutal. Il se produit uniquement quand le test de la boucle est atteint.

### 7.6.2 Attente de la fin d'un thread

Pour attendre qu'un thread se termine, nous disposons de la méthode join :

void **join** ()
: t.join() bloque le thread courant (qui n'est *pas* le thread t), jusqu'à ce que le thread t soit terminé. Notez que si t est déjà terminé, join() revient immédiatement. join() se termine aussi immédiatement si le thread t n'est pas encore démarré.

Cette méthode peut s'avérer utile dans un certain nombre de cas. Supposons par exemple que nous écrivions un jeu en réseau. Chaque joueur s'identifie, et quand l'identification est terminée, le jeux peut commencer. Une première version de l'algorithme serait :

```
pour j= 1 jusqu'à nbrJoueurs
  identifier(j);
commencerJeux();
```

Cependant, cette version impose aux joueurs des contraintes artificielles : le joueur 0 s'identifie forcément avant le joueur 1, qui lui-même s'identifie forcément avant le joueur 2, etc... Si l'identification demande une intéraction forte entre les clients et le serveur, cette solution n'est pas intéressante.

On pourrait donc faire identifier chacun des joueurs en parallèle dans des threads :

```
ThreadIdent t[]= new ThreadIdent[nbrJoueurs];
pour j= 1 jusqu'à nbrJoueurs
      t[i]= new ThreadIdent(i);
      t[i].start();
```

Mais alors, il est nécessaire pour commencer à jouer d'attendre que tous les threads de t soient terminés :

```
pour j= 1 jusqu'à nbrJoueurs
      t[i].join();
```

### 7.6.3 Méthode sleep

La méthode `sleep` endort le thread courant pour un certain nombre de millisecondes. Il existe une méthode `interrupt` qui réveille un thread endormi. Quand cela se produit, la méthode `sleep` renvoie une `InterruptedException`.

La méthode `sleep` peut être intéressante dans des contextes où on ne gère qu'un seul `thread`. Dans ce cas, on l'applique au thread courant en utilisant la ligne :

```
Thread.sleep(1000);
```

`static void` **`sleep`** `(int millis)`
`throws` **`InterruptedException`**

endort le thread courant pour `millis` millisecondes.

`void` **`interrupt`** `()`

interrompt le sommeil d'un thread (en wait() ou en sleep()).

## 7.7 Éviter les attentes actives avec `wait` et `notify`

### 7.7.1 Introduction

Supposons que nous ayons un système multitâche, par exemple un serveur web. Le système est composé de deux types de `threads` :

– un thread serveur, dont le code est :

```
tant que vrai
  si requete.existe()
    requete.executer()
```

– et *des* threads clients dont le code crée des requêtes.

Le problème est le suivant : le code du serveur effectue ici ce que l'on appelle une *attente active,* c'est-à-dire qu'il passe son temps à tester s'il doit travailler. Or cette boucle consomme beaucoup de temps CPU, et ce alors même que le `thread` est sensé *ne rien faire,* ce qui est tout de même paradoxal.

La combinaison des méthodes `wait` et `notify` permet de résoudre le problème. Pour simplifier, disons que `wait` permet de bloquer un thread jusqu'à ce qu'un autre thread le réveille avec `notify()`.

### 7.7.2 Méthodes

`void` **`wait`** `()`
`throws` **`InterruptedException`**

méthode de la classe `Object`. Pour appeler `x.wait()`, il faut se trouver dans une méthode ou un bloc synchronisé sur l'objet `x`.

Le thread courant se bloque, en attendant d'être reveillé par un appel de `notify` ou `notifyAll`. D'autre part, le verrou qu'il détenait sur `x` est relâché, ce qui permet à d'autres threads synchronisés sur `x` de travailler.

`void` **`notify`** `()`
`throws` **`IllegalMonitorStateException`**

`x.notify()` réveille *un* des threads qui mis en attente par `x.wait()`. Le choix effectué est arbitraire. Notez que cela ne signifie pas que le thread en question va forcément avoir la main tout de suite. Si aucun thread n'est en attente, `notify` ne fait *rien.*

`void` **`notifyAll`** `()`
`throws` **`IllegalMonitorStateException`**

notify() est intéressant quand les threads à réveiller sont plus ou moins équivalents. Si un seul des threads en attente sur un wait() est susceptible de pouvoir travailler, rien ne garantit que `notify` réveillera précisément celui-là. Dans ce cas, `notifyAll` peut s'avérer un bon choix. `x.notifyAll()` réveille *tous* les threads en attente sur `x`

**Attention !** Ne confondez pas les threads en attente par la suite d'un `wait()` et les threads qui essaient d'obtenir un verrou pour commencer une fonction synchronized. `notify()` n'aura aucun effet sur ces derniers.

### 7.7.3 Utilisation de wait et notify

Typiquement, `wait` et `notify` s'utilisent comme suit : on a un thread serveur, et un ou plusieurs threads clients. Tous partagent un même objet. L'idée est que quand un client ajoute des données dans l'objet, le serveur doit se mettre à travailler.

Supposons donc que l'objet soit de classe `Compteur`, avec les méthodes :

```
boolean estInitialisé();
void initialise(int v);
// get renvoie la valeur et fixe estInitialisé à faux.
int get();
```

Une première version du serveur pourrait être :

```
while (true) {
  if(compteur.estInitialisé()) {
    int v= compteur.getValeur();
    // On travaille....
  }
}
```

Mais ce code correspond à une *attente active.*

En fait, le code correct sera, pour le serveur :

serveur sans attente active

```
while (true) {
  synchronized (requete) {
```

```
      // On attend qu'il se passe quelque chose sur
      // requete (qu'un client appelle notify())
      while (requete.estVide())
        requete.wait();
      v= requete.getValeur(); // Vide la requête
    }
    // On travaille....
  }
```

Le code du client sera :

client

```
  ...
  synchronized (requete) {
    if (requete.estVide()) {
      requete.initialise(1);
      requete.notify();
    }
  }
  ...
```

Comment cela fonctionne-t-il ? Lorsque le serveur se lance, il prend le verrou sur `compteur`. Puis il rentre dans la boucle. Si le compteur n'a pas de valeur, le serveur se met en attente et relâche le verrou. Lorsqu'un client initialise le compteur, il effectue ensuite un `notify`, qui interrompt le `wait`. Le serveur peut à nouveau tourner[2]. Lorsqu'il a la main, il reprend son exécution, et comme `estInitialisé` est vrai, il exécute le code qui suit.

### 7.7.4 Client et serveur sans attente active

Dans l'exemple précédent, le client essaie d'envoyer une requête, et, si ce n'est pas possible, passe à autre chose. Seul le serveur attend. Il est possible de faire attendre les clients comme le serveur. Le problème qui se pose alors est que `notify` ne prévient qu'un seul *thread.* Il n'est pas garanti que ce soit le bon. Par exemple, si un client pose une requête, le notify peut très bien réveiller, non le serveur, mais un autre client, ce qui ne sert à rien. Pour résoudre ce problème, il est possible d'utiliser `notifyAll`, qui réveille *tous* les threads qui effectuent un `wait` sur une variable donnée.

Le code du serveur devient alors :

serveur sans attente active

```
while (true) {
  synchronized (requete) {
    // On attend qu'il se passe quelque chose sur
    // requete (qu'un client appelle notifyAll())
    while (requete.estVide())
      requete.wait();
    v= requete.getValeur(); // Vide la requête
```

2. Mais cela ne signifie pas qu'il reprenne la main immédiatement. Il est toujours en concurrence avec les autres threads. Pour que le serveur ait immédiatement la main, il suffirait néanmoins de lui donner une priorité supérieure à celle des clients.

```
    requete.notifyAll(); // Prévient tout le monde que requête est modifiée
  }
  // On travaille....
}
```

et le code du client :

client
```
// On travaille ...
synchronized (requete) {
  // On attend que la requête soit libre :
  while (! requete.estVide())
    requete.wait();
  requete.initialise(1);
  requete.notifyAll();
}
// On travaille ...
```

Lorsqu'un client dépose une requête, il prévient le serveur et les autres clients en attente. Mais ce n'est pas gênant : le serveur va pouvoir travailler, et les autres clients, comme la requête est occupée, vont se remettre en `wait`.

### 7.7.5 Gestion d'une file de messages

L'exemple `PileAttente.java` correspond à un cas fréquent : le serveur reçoit une liste de messages qu'il doit traiter, de préférence dans l'ordre d'arrivée.

PileAttente.java
```
/**
 * Démo d'un programme de gestion d'événements.
 * On a l'architecture suivante :
 * n Threads clients qui envoient des événements à un serveur
 * un Thread qui gère la pile des événements
 * un thread qui effectue les traitements
 *
 * Le but de ce programme d'exemple est d'éviter une
 * attente active de la part du serveur.
 * En effet, l'algorithme le plus simple pour le serveur serait :
 * while (true) {
 *   if (! pile.empty())
 *     v= depiler()
 *     traiter(v);
 *   yield();
 * }
 *
 * qui consomme beaucoup de CPU pour rien.
 */

import java.util.*;

class Client extends Thread {
```

```
    PileAttente serveur;
    Random attente;
    int id;
    int temps;

    public Client(PileAttente serveur, int id, int temps, Random r) {
        this.serveur=serveur;
        this.id= id;
        this.temps= temps;
        attente= r;
    }

    public void run() {
        while(true) {
            try {
                sleep(attente.nextInt(1000* 2* temps));
            } catch (InterruptedException e) {}

            System.out.println("Le Thread " + id + " va empiler " + temps);
            serveur.empiler(temps);
            System.out.println("Le Thread " + id + " a empilé " + temps);
            // yield();
        }
    }

}

public class PileAttente extends Thread {

    /**
     * traiter attends val secondes, puis affiche val.
     */
    Stack pile;

    public PileAttente() {
        pile= new Stack();
    }

    public void traiter(int val) {
        System.out.println("On traite");
        try {
            sleep(val* 1000);
        } catch (InterruptedException e) {};
        System.out.println("Fin du traitement "+ val);
    }

    public void run() {
        while (true) {
            System.out.println("Le serveur veut lire une valeur ");
            int x= depiler();
            System.out.println("Le serveur a lu " + x);
```

```
            traiter(x);
        }
    }

    public synchronized void empiler(int v) {
        pile.push(new Integer(v));
        notify();
        // On pourrait mettre aussi notifyAll, en prévision du cas où
        // l’on aurait plusieurs *serveurs*.  Mais notifyAll serait-il
        // intéressant ? Si plusieurs threads sont en attente pour
        // dépiler, ils sont équivalents. Il suffit donc d’en prévenir un.
        // d’où notify.
    }

    public synchronized int depiler() {
        int v;
        while (pile.empty()) {
            try {
                wait(); // On attends un notify.
            } catch (InterruptedException e) {};
        }
        System.out.println("Le serveur est réveillé et va lire");
        v= ((Integer)pile.pop()).intValue();
        return v;
    }

    public static void main(String args[]) {
        PileAttente s= new PileAttente();
        Random r= new Random();

        Client c1= new Client(s, 1, 5, r);
        Client c2= new Client(s, 2, 7, r);
        Client c3= new Client(s, 3, 10, r);
        s.start();
        c1.start();
        c2.start();
        c3.start();
    }
}
```

## 7.8 Démons

En première approximation, un programme java s’arrête quand tous les threads se sont arrêtés. Or, certains threads servent uniquement à fournir des services aux autres ; leur code a généralement la forme :

```
public void run() {
  while (true) {
```

```
    si requete
        traiter_requete();
    sleep(1000)
  }
}
```

Un thread avec un tel `run()` ne s'arrêterait jamais. Pour simplifier la vie du programmeur, Java introduit la notion de démon. La machine virtuelle java s'arrête en fait quand les seuls threads qui tournent sont des démons.

on déclare un thread comme démon grâce à la méthode `setDaemon` :

`public void` **`setDaemon`** `(boolean d)`
: déclare le thread comme un démon si `d` est `true.` Attention, ne peut être utilisée sur un thread une fois qu'il est lancé.

## 7.9 Graphismes

Quand un programme graphique est lancé, java démarre un thread, la *boucle graphique,* qui va :

- recevoir les événements graphiques (clics de souris, fenêtre visible...) ;
- les traiter (en appelant les *listeners* correspondants) ;
- s'occuper de l'affichage.

En conséquence, si un programme graphique effectue un traitement long, l'intégralité du programme est gelé pendant ce temps : pas d'affichage, ni de réponse aux actions de l'utilisateur.

Plusieurs solutions sont disponibles : l'utilisation d'un `Timer` swing (en particulier quand on veut une animation), le package spin (`http://spin.sourceforge.net`), et les threads.

L'idée est de placer l'opération longue dans un thread. Ainsi, l'interface graphique se contentera de mettre en place le thread, et de le lancer.

Rebonds.java

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

// Observateur
public class Rebonds extends JPanel {
    Billard billard;

    public Rebonds (Billard b) {
        billard= b;
    }

    public void paintComponent(Graphics g) {
        // Dessiner le fond. Obligatoire pour un bon fonctionnement !
```

```
        super.paintComponent(g);
        int x, y;
        // Erreur possible dans les deux lignes suivantes !
        // Laquelle ?
        x= billard.getX();
        y= billard.getY();
        x= (x * getWidth()) / billard.WIDTH;
        y= (y * getHeight()) / billard.HEIGHT;
        g.fillOval(x - 3, y - 3, 6, 6);
    }

    public static void main(String args[]) {
        Billard b= new Billard(10, 31);
        Rebonds r= new Rebonds(b);
        Mouvement mouvement= new Mouvement(r, b);
        JFrame jf= new JFrame();
        jf.getContentPane().add(r);
        jf.setSize(200,200);
        jf.setVisible(true);
        jf.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }});
        mouvement.setPriority(10);
        mouvement.start();

    }
}

// Modèle
class Billard {
    // Position de la balle
    private int x,y;

    public final static int WIDTH= 100;
    public final static int HEIGHT= 100;

    public Billard(int x, int y) {setPos(x,y);}
    synchronized public int getX() {return x;}
    synchronized public int getY() {return y;}

    synchronized public void setPos(int x, int y) {this.x= x; this.y= y;}
}

// Action
class Mouvement extends Thread {
    Rebonds rebonds;
    Billard billard;
    // Direction de la balle
    int dx;
    int dy;
```

```
    public Mouvement(Rebonds rebonds, Billard billard) {
        this.rebonds= rebonds;
        this.billard= billard;
        dx= 1;
        dy= 2;
    }

    public void run() {
        while (true) {
            int x= billard.getX();
            int y= billard.getY();
            if (x + dx < 0 || x + dx > billard.HEIGHT)
                dx= -dx;
            if (y + dy < 0 || y + dy > billard.HEIGHT)
                dy= -dy;
            billard.setPos(x+dx, y+dy);
            // se rappeler de prévenir rebonds !!!
            rebonds.repaint();
            yield();
            try {
                sleep(0,01);
            }  catch (InterruptedException e) {}
        }
    }
}
```

Le code qui précède montre l'utilisation d'un thread, en l'occurrence pour lancer une animation. Mais c'est valable pour tout traitement un peu long.

L'erreur évoquée dans le code est la suivante : si le thread modifie la position du point entre l'exécution de `x= billard.getX();` et celle de `y= billard.getY();`, la balle sera affichée à une position fausse. La solution serait de fournir une méthode `getPosition` synchronisée.

### 7.9.1 synchronisation et graphismes

La plupart des méthodes de SWING ne sont pas synchronisées. *A priori*, seul la boucle graphique devrait donc manipuler les objets swing. Certaines méthodes sont cependant sûres : on peut ainsi utiliser `repaint` sans problème.

Pour les autres manipulations :

- utiliser `SwingUtilities.invokeLater(Runnable)`
- ou utiliser `SwingUtilities.invokeAndWait(Runnable)`

qui insèrent les actions dans la liste d'événements de SWING.

# Deuxième partie

# Bibliothèques particulières

# Chapitre 1

# Les Collections

## 1.1 Introduction

Les *collections* sont un ensemble de classes de java qui permettent d'organiser des données en tableaux, ensembles, dictionnaires, etc... les *collections* fournissent de plus une séparation entre leur implémentation effective et leur usage, permettant ainsi une meilleure réutilisabilité. Pour cela, on utilise l'héritage et les interfaces.

Voici tout d'abord un exemple de code :

```
1  import java.util.*;
2
3  public class TestCollec {
4      public static void main(String argc[])
5      {
6          Set promotion= new TreeSet();
7          promotion.add(new Etudiant(1, "Turing"));
8          promotion.add(new Etudiant(2, "Babbage"));
9          if (promotion.contains(new Etudiant(4, "Wirth")))
10             System.out.println("Wirth appartient à la promotion");
11     }
12 }
```

ligne 1 les collections sont dans le package `java.util`

ligne 6 on crée une collection de *classe* `TreeSet`, et on la range dans un handle vers un `Set`. Nous reviendrons sur cette ligne.

ligne 7 On ajoute un nouvel étudiant dans l'ensemble « `promotion` », à l'aide de la méthode `add`.

ligne 9 On teste si Wirth appartient à la promotion, grâce à la méthode `contains`.

La ligne 6 est très importante. Ce que l'on veut, dans la suite du programme, c'est un ensemble — et en java, cela signifie « un objet qui implémente l'interface `Set` ». Peu importe,

pour la suite, la manière dont cet ensemble est implémenté. Par contre, lors de la création de cet ensemble, nous sommes bien forcé de choisir un implémentation effective. Nous choisissons ici `TreeSet` (implémentation d'un ensemble grâce à un arbre).

Un point important est que nous ne précisons nulle part que `TreeSet` est un ensemble d'étudiants. En fait, `TreeSet` est un ensemble d'`Object` (pour ce type particulier d'ensemble, on a de plus l'obligation que les objets en question implémentent l'interface `Comparable`, c'est-à-dire qu'ils possèdent une méthode `compareTo`).

## 1.2 Itérateurs

Supposons que nous ayons rangé des etudiants dans un tableau. L'affichage de l'ensemble des étudiants se ferait de la manière suivante :

```
for (int i=0; i< tabEtuds.length; i++)
        System.out.println(tabEtuds[i]);
```

Supposons que nous ayons créé un type « liste d'étudiants ». Nous le parcourrions de la manière suivante :

```
for (ListeEtud l= listeEtuds; l != null; l= l.next)
        System.out.println(l.getEtud());
```

Du point de vue de *l'implémentation,* ces deux approches sont très différentes. Par contre, du point de vue de l'utilisation, on remarque que `l` et `i` sont similaires. Ils disposent chacun des fonctionnalités suivantes :

- on les initialise au « début » de l'ensemble (`i=0`, `l= listeEtuds`) ;
- on dispose d'un opérateur pour passer à l'élément suivant : (`i++` et `l= l.next`) ;
- on dispose d'un test d'arrêt : `i < tabEtuds.length` et `l != null` ;
- on dispose d'une opération pour récupérer l'élément courant : tabEtuds[i] et `l.getEtud()`.

De plus, ces quatre opérations permettent conceptuellement de parcourir n'importe quel ensemble.

En java, ce concept est actualisé par l'interface `Iterator`. Le code précédent peut alors s'écrire, pour n'importe quelle collection, de la manière suivante :

```
for (Iterator i= promotion.iterator(); i.hasNext(); )
{
        Etudiant e= (Etudiant) i.next(); [1]
        System.out.println(e);
}
```

Remarquez que c'est `promotion` qui construit l'itérateur. C'est encore un usage de l'héritage !

Les opérations vues précédemment étant représentées comme suit :

**Création, initialisation** La création d'un itérateur est effectuée par la collection elle-même, par l'appel de la méthode `iterator()` ;

**Récupération de l'élément courant *et* avancement** La méthode `next()` renvoie la valeur de l'élément courant, puis avance d'un cran (cf. `i++` en C).

**Test d'arrêt** La méthode `hasNext` retourne vrai s'il est possible d'appeler `next()`

## 1.3 Organisation des classes

L'organisation de l'arbre d'héritage pour les collections est le suivant :

Les *interfaces* définissent les grandes lignes de ce que peut faire une collection :

```
interface java.util.Collection
    interface java.util.List
    interface java.util.Set
          interface java.util.SortedSet
interface java.util.Map
    interface java.util.SortedMap
interface java.util.Map.Entry
```

Les classes sont organisées comme suit :

```
class java.util.AbstractCollection (implements java.util.Collection)
  class java.util.AbstractList (implements java.util.List)
    class java.util.AbstractSequentialList
      class java.util.LinkedList (implements java.util.List)
    class java.util.ArrayList (implements java.util.List)
    class java.util.Vector (implements java.util.List)
      class java.util.Stack
  class java.util.AbstractSet (implements java.util.Set)
    class java.util.HashSet (implements  java.util.Set)
    class java.util.TreeSet (implements  java.util.SortedSet)
 class java.util.AbstractMap (implements java.util.Map)
  class java.util.HashMap (implements  java.util.Map)
  class java.util.TreeMap (implements  java.util.SortedMap)
  class java.util.WeakHashMap (implements java.util.Map)
```

Donnons les clefs pour comprendre ces interfaces et ces classes.

Tout d'abord, conceptuellement :

- les *listes* (`List`) sont des suites d'éléments. On peut, soit les parcourir du premier au dernier, soit accéder au i$^{e}$ élément ;

---

1. On remarquera un problème de conception dû à la reprise d'habitudes provenant du C. Il serait bien plus souple et plus général de couper `next()` en *deux* opérations : une qui avance dans la liste, l'autre qui permet de récupérer un élément.

– les *ensembles* (Set) ne comportent pas de doublets. Un élément appartient à un ensemble ou ne lui appartient pas. Contrairement aux listes, on ne peux pas choisir *où* insérer un élément. Les SortedSet sont des ensembles triés, c'est-à-dire que les éléments seront rangés du plus petit au plus grand (selon la méthode compareTo).

– les dictionnaires (Map) sont des tableaux associatifs. C'est-à-dire des tableaux dont les indices sont d'un type quelconque. On peut par exemple avoir comme indice une chaîne de caractères, et écrire des choses comme :

```
definitions.put("chauve-souris", "mammifère insectivore volant");
```

La dernière interface, Map.Entry, permet de manipuler un dictionnaire comme un ensemble (Set) de définitions.

En ce qui concerne les classes, elles fournissent des implémentations de ces interfaces. Le choix de la classe utilisée dépend de l'usage que l'on veut en faire [2].

– LinkedList fournit une liste doublement chaînée. L'ajout d'un élément est rapide, mais la recherche d'une valeur donnée et l'accès au i^e^ élément sont en $\mathcal{O}(n)$.

– ArrayList est une implémentation à base de tableau. L'ajout d'un élément peut être plus coûteux que dans le cas d'une LinkedList si le tableau doit grossir, mais par contre l'accès au i^e^ élément est en temps constant.

– Vector est un équivalent de ArrayList datant du jdk1.0. Les différences entre les deux se situent surtout lorsque l'on utilise le multitâche [3].

– Stack est un type spécial de vercteur, utilisé pour réaliser des *piles*. Les piles sont des structures de données très utiles, dont nous reparlerons.

– Pour le reste :

  – Hash signifie que l'implémentation utilise un algorithme de hachage : elle associe à chaque élément un nombre (par exemple, on pourrait associer à une chaîne de caractère la somme des codes des caractères qu'elle contient), et elle utilise ce nombre comme indice dans un tableau. Les implémentations par hachage sont généralement très rapides pour des volumes moyens de données. Par contre, elles ne permettent pas de récupérer les éléments dans l'ordre.

  – Tree signifie que l'implémentation utilise un arbre. Les arbres sont des structures très robustes, adaptées à de grands volumes de données. De plus, ils permettent de récupérer les éléments dans l'ordre croissant.

2. Nous ne parlerons pas ici des WeakHashMap, qui sont trop techniques pour cette première approche
3. Pour être complet, disons que Vector est synchronisé. Pour comprendre, attendre de voir les Threads

## 1.4 L'interface Collection

Cette interface explicite les fonctionnalités communes à toutes les collections.

```
public interface Collection {
    // Opérations de base
    int size(); // Nombre d'éléments
    boolean isEmpty();  // la collection est-elle vide ?
    void clear(); // vide la collection

    // Opérations sur les éléments
    boolean contains(Object element); // la collection contient-elle element
    boolean add(Object element);   // ajout d'un élément (optionnel)
    boolean remove(Object element); // suppression d'un élément (optionnel)

    // Itérateur
    Iterator iterator(); // récupère un itérateur sur le premier élément

    // Opération inter-collections
    boolean containsAll(Collection c); // la collection inclut-elle c ?
    boolean addAll(Collection c); // this = this ∪ c
    boolean removeAll(Collection c); // this = this − c
    boolean retainAll(Collection c); // this = this ∩ c


    // tableaux :
    Object[] toArray(); // Renvoie le tableau des éléments de la collection
    Object[] toArray(Object a[]);
}
```

**Notes :** `add, addAll`, etc... sont des méthodes de Collection qui retournent un booléen, qui précise si la méthode a modifié l'ensemble auquel elle s'applique. Par exemple, si $X = \{a, b, c\}$ et que $Y = \{b, c\}$, `X.addAll(Y)` ne modifiera pas X, puisque tous les éléments de Y sont dans X. Donc `X.addAll(Y)` renverra `false`. Par contre, `Y.addAll(X)` renverrait `true`.

Les méthodes qui doivent chercher un élément (`contains, remove`...) utilisent les méthodes `equals` des éléments pour les comparer entre eux.

La dernière version de `toArray` permet en fait de récupérer un tableau typé. On peut écrire, par exemple,

```
String[] x = (String[]) maCollectionDeChaines.toArray(new String[0]);
```

pour récupérer dans `x` un tableau de `String`.

Les itérateurs fournissent les opérations :

```
public interface Iterator {
    boolean hasNext(); // Peut-on appeler next ?
    Object next(); // Retourne l'élément courant et incrémente
    void remove(); // Enlève l'élément courant (opt)
}
```

L'opération `remove` est optionnelle, c'est-à-dire que certaines implémentation de l'interface peuvent ne pas en donner de version utilisable. Il faut consulter la documentation.

## 1.5 Listes

Les listes sont des suites d'éléments. L'ordre des éléments est donc important, et est fixé par le programmeur. Les opérations de `List` sont les mêmes que celles de `Collection`, plus :

```
public interface List extends Collection {
    // Accès direct
    Object get(int i);\ Récupère le ie élément
    Object set(int i, Object elt); // fixe le ie elt. Opt.
    void add(int i, Object elt); // Ajoute elt en ie position
                                 // Déplace les suivants vers la droite
    Object remove(int i); // enlève le ième elt
                          // Déplace les suivants vers la gauche
    abstract boolean addAll(int i, Collection c);
                                     // ajoute en pos. i les elts de c

    // Recherche
    int indexOf(Object o); // Index de la première occurrence de l'objet o, ou -1
    int lastIndexOf(Object o); // Index de la dernière occurrence de l'objet o, ou -1

    // Iterateurs
    ListIterator listIterator(); // Itérateur de liste
    ListIterator listIterator(int i); // Itérateur de liste sur le ie élément

    // Sous listes
    List subList(int from, int to); // sous liste entre from inclus et
                                    // to, exclu
}
```

Outre ces opérations, les listes fournissent des itérateurs de liste. Ceux-ci sont dotés des opérations supplémentaires :

```
public interface ListIterator extends Iterator {
    boolean hasPrevious(); // Passage au précédent possible ?
    Object previous(); // passage à l'objet précédent

    int nextIndex(); // Index suivant
    int previousIndex(); // Index précédent

    void set(Object o); // fixe l'élément courant à o (opt)
    void add(Object o); // ajoute l'élément o à cette position
}
```

Remarque sur les opération de recherche : `contains, remove, retainAll` sont spécifiées pour les liste en utilisant l'opérateur `equals` de l'élément, pas l'opérateur ==. Notez par contre que les opérations d'ajout rajoutent un élément, même s'il se trouve déjà dans la liste, et

que si `remove(Object)` supprime la première occurrence de l'objet, `c1.removeAll(c2)` garantit qu'après exécution, il n'y a plus aucun élément contenu dans c2 qui reste dans c1.

## 1.6 Ensembles

L'interface d'ensemble est la même que celle de Collection. Il est par contre important de noter les points suivants sur les implémentations.

### 1.6.1 Ensembles triés

Pour construire un ensemble trié, il faut expliquer *comment* trier ces éléments. C'est automatique si les classes des éléments implémentent l'interface `Comparable`. Cette interface définit une méthode `compareTo` :

```
public int compareTo(Object o)
```

Cette méthode renvoie un entier négatif, null, ou positif selon que `this` est (respectivement) inférieur, égal, ou supérieur à `o`.

L'autre solution est de fournir un *comparateur* comme argument au constructeur de l'ensemble.

Un comparateur est un objet d'une classe qui définit la méthode suivante [4] :

```
int compare(Object o1, Object o2); // Compare o1 et o2 (cf. ci-dessus)
                                   // Pour des entier, o1 - o2 fait l'affaire
```

Par exemple, ce comparateur permet de créer un ensemble d'entiers triés en ordre décroissant :

```
class MonComparateur implements Comparator {
    public int compare(Object o1, Object o2)
    {
        Integer a= (Integer)o1; Integer b= (Integer)o2;
      return b.intValue()-a.intValue();
    }
}
```

On l'utilisera ainsi :

```
Set s= new TreeSet(new MonComparateur());
s.add(new Integer(5));
s.add(new Integer(7));
for (Iterator i= s.iterator(); i.hasNext(); )
   System.out.println(i.next());
```

Ce qui imprime : « 7 5 »

Notez l'usage de `Integer` pour encapsuler un `int` dans un objet.

4. Ce n'est pas l'exacte vérité mais c'est suffisant en pratique. Pour tout savoir, lire la doc !

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com



## 1.7 Dictionnaires (maps)

```
public interface Map {
    // Operations de base
    // Range la valeur v dans la case k
    Object put(Object k, Object v);
    // Récupère la valeur v associée à k
    Object get(Object k);
    // Supprime la case de clé k
    Object remove(Object k);
    // possède-t-on un case de clé k ?
    boolean containsKey(Object k);
    // possède-t-on un case contenant la valeur v ?
    boolean containsValue(Object v);
    int size();
    boolean isEmpty();

    // Opérations globales
    // Range tous les éléments de t dans this
    void putAll(Map t);
    void clear();

    // Vue comme collections
    // Ensemble des clefs
    public Set keySet();
    // Ensemble des valeurs
    public Collection values();
    // Ensemble des couple clef/valeurs
    public Set entrySet();

    // Interface pour les éléments de entrySet
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```

Notez l'absence d'`iterator()`. En fait, pour parcourir une `Map`, on passe par les ensembles (`Set`), en utilisant, l'une des trois méthodes `keySet()`, `values()`, ou `entrySet()`, selon que l'on veut parcourir l'ensemble des clefs, des valeurs, ou des couples clefs/valeurs.

On utilise cela de la manière suivante :

```
Set s= dico.entrySet();
for(iterator i= s.iterator(); i.hasNext(); )
  {
    // Map.Entry (notez l'orthographe, il s'agit d'une sous-classe)
    // représente un couple clef/valeur dans la Map.
    Map.Entry e= (Map.Entry)(i.next());
    System.out.println("clef " + e.getKey() + ", valeur : ", e.getValue());
  }
```

## 1.8 Piles (`Stack`)

Une pile est un objet doté des opérations :

```
boolean empty(); // est-elle vide ?
void push(Object o); // empile un objet
Object pop(); // dépile un objet
Object peek(); // regarde le sommet de la pile
```

## 1.9 Exemples

### 1.9.1 La collection Sac

Nous présentons ici un exemple « jouet » de collection, pour montrer, en particulier, le lien entre une collection et son itérateur.

```
1  package mesutils;
2  import java.util.*;
3
4  public class Sac extends AbstractCollection {
5      int utilisees; // nombre de cases utilisées
6      Object tab[]; // tableau des objets de la collection
7
8      // Constructeur
9      public Sac(int capacite) {
10         utilisees=0;
11         tab= new Object[capacite];
12     }
13
14     public int size() {
15         return utilisees;
16     }
17
18     public boolean add(Object o) {
19         tab[utilisees++]= o;
20         return true; // On retourne vrai si la collection a changé
21     }
22
23     public Iterator iterator() {
24         return new SacIterator(this);
25     }
26
27     // Test et démonstration
28     public static void main(String args[]) {
29         Collection c= new Sac(10);
30         c.add("I"); c.add("II"); c.add("III"); c.add("IV"); c.add("V");
31         c.add("VI"); c.add("VII"); c.add("VIII"); c.add("IX"); c.add("X");
32         for (Iterator i= c.iterator(); i.hasNext(); )
33             System.out.println("élément " + i.next());
```

```
34          // erreur :
35          c.add("XI");
36      }
37  }
```

```
1   package mesutils;
2   import java.util.*;
3
4   public class SacIterator implements Iterator {
5       int i;
6       Sac s;
7
8       // Seul Sac a le droit d'appeler le constructeur
9       // Celui-ci a donc la portée ''package''.
10      SacIterator(Sac s) {
11          i= 0;
12          this.s= s;
13      }
14      public boolean hasNext() {
15          return i < s.size();
16      }
17
18      public Object next() {
19          return s.tab[i++];
20      }
21
22      public void remove() {
23          throw new
24              java.lang.UnsupportedOperationException(
25                  "Remove n'est pas implémentée pour les sacs");
26      }
27  }
```

### 1.9.2 Utilisation de Map pour compter des mots

Le programme suivant lit un fichier, ligne par ligne, et affiche ensuite le nombre de fois où chaque ligne apparaît.

```
1   import java.util.*;
2   import java.io.*;
3
4   public class TestMaps {
5       public static void main(String args[]) throws IOException {
6           BufferedReader f= new BufferedReader(new InputStreamReader(System.in));
7           String s;
8           Map dico= new TreeMap(); // Dictionnaire trié
9           while ((s= f.readLine()) != null) {
10              int nb= 0;
11              if (dico.containsKey(s))
12                  nb= ((Integer)dico.get(s)).intValue();
```

```
13              // nb contient maintenant le nombre d'occurrences du mot.
14              dico.put(s, new Integer(nb + 1));
15          }
16          f.close();
17          // Affichage :
18          Set entrees= dico.entrySet();
19          for(Iterator i= entrees.iterator(); i.hasNext();)
20              {
21                  Map.Entry e= (Map.Entry)(i.next());
22                  System.out.println(e.getKey() + " apparaît " +
23                                      e.getValue() + " fois");
24              }
25      }
26  }
```

Voici un exemple d'utilisation :

```
[rosmord@djedefhor Cours]$ java TestMaps
aa
ppp
aa
ll
l
mmm
l
aa
b
o
a (Contrôle-D)
a apparaît 1 fois
aa apparaît 3 fois
b apparaît 1 fois
l apparaît 2 fois
ll apparaît 1 fois
mmm apparaît 1 fois
o apparaît 1 fois
ppp apparaît 1 fois
```

# Chapitre 2

# Entrées/sorties

**INTÉGRER LES EXEMPLES DES TRANSPARENTS**

## 2.1 La classe File

La classe File permet de manipuler des fichiers et des répertoires de l'extérieur : elle fournit les fonctionalités nécessaires pour connaître les droits sur un fichier, le détruire, le déplacer, lister des répertoires, etc... Par contre elle ne fournit rien pour écrire dedans.

Pour voir ce que permet la classe File, voici un petit programme qui détruit les fichiers de sauvegardes créés par emacs (leur nom se termine par « ~ »).

Clean.java

```
1  import java.io.*;
2
3  // FilenameFilter est une interface utilisée par listFiles pour
4  // ne lister qu'une partie des fichiers. Chaque fichier du répertoire
5  // est testé par la méthode accept(...), qui renvoie true si le fichier
6  // doit apparaître. Ici, nous vérifions que le nom du fichier se termine
7  // par "~".
8
9  public class Clean implements FilenameFilter {
10
11     public boolean accept(File dir, String name) {
12         return (name.charAt(name.length() -1) == '~');
13     }
14
15     static public void main(String args[]) {
16         // On crée un filtre pour listFiles.
17         // Comme il s'agit d'un bête petit programme,
18         // on utilise la classe Clean comme filtre.
19         Clean filter= new Clean();
20         // dir représente le répertoire à nettoyer.
21         // Son nom est passé comme argument.
22         File dir= new File (args[0]);
23         if (dir.isDirectory()) {
```

```
24              // On crée la liste des fichiers
25              // dont le nom se termine par ~
26              File [] fichs= dir.listFiles(filter);
27              // On efface tous ces fichiers.
28              for (int i= 0; i< fichs.length; i++)
29                  fichs[i].delete();
30          }
31      }
32  }
```

Nous donnons ci-après une sélections de méthodes utiles fournies par la classe File.

`public` **`File`** `(String pathname)`
: Crée un nouvel objet `File`, correspondant au chemin (path) indiqué. Par exemple, avec

```
File f= new File(".");
```

`f` désignera le répertoire courant. Le chemin peut correspondre à un fichier ou à un répertoire.

`public` **`File`** `(File parent, String child)`
: Crée un nouvel objet `File`, correspondant au chemin (path) composé en ajoutant `child` à `parent`. Ainsi, en exécutant le code :

```
File homedir= new File("/home/rosmord");
File f= new File(homedir, "MonProg.java");
```

`f` désignera le fichier "`/home/java/MonProg.java`".

`boolean` **`canRead`** `()`
: Renvoie vrai si le fichier est lisible.

`boolean` **`canWrite`** `()`
: Renvoie vrai si le fichier est autorisé en écriture.

`boolean` **`delete`** `()`
: Détruit le fichier ou le répertoire correspondant.

`boolean` **`exists`** `()`
: Renvoie vrai si le fichier existe.

`String` **`getName`** `()`
: Retourne le nom du fichier.

`String` **`getPath`** `()`
: Retourne le chemin du fichier (nom compris).

`String` **`toString`** `()`
: même chose que `getPath`.

`boolean` **`isDirectory`** `()`
: Renvoie vrai si le fichier est un répertoire.

`boolean` **`isFile`** `()`
: Renvoie vrai si le fichier est un fichier normal.

`long` **`lastModified`** `()`
: Retourne la date de dernière modification.

`long` **`length`** `()`
: Retourne la longueur du fichier

`File[]` **`listFiles`** `()`
: Renvoie la liste des fichiers contenus dans `this,` qui doit bien entendu être un répertoire.

`File[]` **`listFiles`** `()`
: Renvoie la liste des fichiers contenus dans `this,` qui doit bien entendu être un répertoire.

`File[]` **`listFiles`** `(FilenameFilter filter)`
: Idem, mais ne concerne que les fichiers sélectionnés par filter. Voir l'exemple de code en introduction.

`boolean` **`mkdir`** `()`
: Crée le répertoire correspondant à `this`.

`boolean` **`mkdirs`** `()`
: `this` désigne un répertoire. Cette méthode le crée, mais crée aussi tous les répertoires intermédiaires nécessaires, s'ils n'existent pas. Par exemple, avec le code :

```
File f= new File("/tmp/rep1/rep2/rep3");
f.mkdirs();
```

Si ni rep1, ni rep2, ni rep3 n'existent, les trois seront créés.

`boolean` **`renameTo`** `(File dest)`
: Renomme un fichier.

`boolean` **`setReadOnly`** `()`
: Place un fichier en lecture seule. (interdit l'écriture, quoi).

## 2.2 Organisation des classes d'entrées/sortie

### 2.2.1 Fichiers en lecture, fichiers en écriture

### 2.2.2 Classes orientées octets, classes orientées caractères

### 2.2.3 Filtrage

Les classes d'entrées/sorties de Java sont souvent utilisées « en cascade ». L'idée étant que le texte lu par un objet sert d'entrée au suivant. On remarquera que ce système est assez similaire à celui qui est utilisé dans les `pipe` d'Unix.

Considérons, par exemple, la ligne :

```
LineNumberReader f= new LineNumberReader(new InputStreamReader(System.in));
```

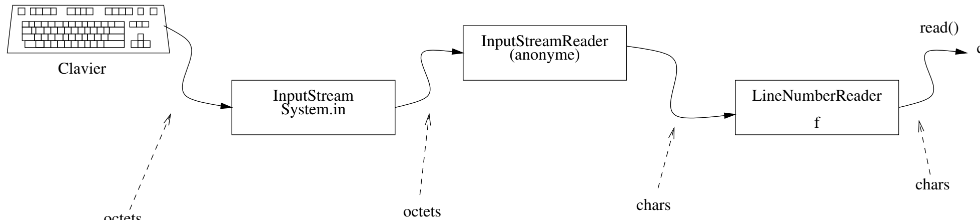
On utilise trois objets :

- `System.in`, qui est un `InputStream`, c'est à dire un fichier ouvert en *lecture*, et lisant des *octets.*
- `InputStreamReader`, qui prend en *entrée,* le flot d'octets fourni par `System.in,` et renvoie des caractères.
- `LineNumberReader` : lit les caractères dans l'`InputStreamReader,` et les renvoie. Au passage, en profite pour tenir à jour un numéro de ligne.

Lorsqu'une instruction comme

```
c= f.read();
```

est exécutée, le cheminement des données est le suivant :




Pour mieux comprendre le fonctionnement de ce système, voyons une petite classe qui fourni les fonctionnalités d'un Reader (Lecteur de caractères), mais compte les caractères lus :

```
import java.io.*;
public class CompteReader extends Reader {
    private Reader in;
    private int total;
    // ''in'' est le Reader dans lequel nous lirons
    public CompteReader(Reader in) {
        this.in= in;
```

```
        total= 0;
    }
    public int read(char[] s, int off, int len) throws IOException
    {
        // On lit dans in, et on récupère le nombre de caractères
        // lus dans nbr.
        int nbr= in.read(s,off,len);
        // On augmente le total en conséquence.
        total+= nbr;
        return nbr;
    }
    public void close() throws IOException
    {
        in.close();
    }
    public int getTotal() {
        return total;
    }
}
```

Quand on lit une donnée dans un `CompteReader`, avec `read()`, la méthode `read` de `CompteReader` utilise celle du `Reader` « `in` » pour lire effectivement les caractères.

## 2.3 Quelques Classes

**`FileReader`, `FileWriter`, `FileInputStream`, `FileOutputStream`** : ces classes lisent et écrivent dans un fichier.

**`BufferedReader`** : cette classe utilise un *tampon* pour lire les données, c'est-à-dire que les données sont lues par blocs, lesquels sont stockés en mémoire. L'avantage de cette méthode est qu'elle réduit normalement le nombre d'accès disque. Son utilisation est transparente pour le programmeur.

Un autre avantage du `BufferedReader` est la possibilité de revenir en arrière dans la lecture, à l'aide des méthodes `mark` et `reset` :

`void` **`reset`** `()`
    `throws` **`IOException`**

remet la tête de lecture à la position de la dernière marque posée.

`void` **`mark`** `(int limite)`
    `throws` **`IOException`**

place une marque à la position courante de lecture. l'argument `limite` permet de fixer le nombre maximum de caractères qui pourront être lus en préservant la marque.

Un autre intérêt du `BufferedReader` est la disponibilité de la méthode `readLine` :

`String` **`readLine`** `()`
    `throws` **`IOException`**

renvoie la ligne suivante sous forme de `String`, ou `null` si on est à la fin du fichier.

**`LineNumberReader:`** un lecteur dérivé de `BufferedReader`, mais doté d'une méthode `getLineNumber` qui permet de connaître le numéro de la ligne courante.

**`InputStreamReader:`** cette classe permet de construire un `Reader` à partir d'un `InputStream`. C'est particulièrement utile avec `System.in`.

**`PushbackReader:`** un lecteur doté d'une opération `unRead()` qui permet de remettre du texte dans le flot d'entrée, comme s'il n'avait pas déjà été lu.

**`CharArrayReader, CharArrayWriter, ByteArrayInputStream, ByteArrayOutputStream, StringReader, StringWriter, StringBufferInputStream:`** ces classes permettent de lire ou d'écrire dans un tableau ou dans une String. Par exemple, le code suivant lit le contenu d'un fichier texte et le copie dans une `String` :

```
int c;
FileReader f= new FileReader(args[0]);
StringWriter sw= new StringWriter();
while ((c= f.read()) != -1) {
  // écrit c dans sw
  sw.write(c);
}
// Copie le contenu de sw dans s.
String s= sw.toString();
System.out.println(s);
```

## 2.4 Exemples d'utilisation

### 2.4.1 Lecture d'un fichier

```
public static void testLectureTexte (InputStream i) throws IOException {
  InputStreamReader f= new InputStreamReader(i);
  // Le résultat de read doit être un entier
  // (à cause du -1 qui est renvoyé en fin de fichier)
  int cc;
  while ( (cc = f.read()) != -1)
  {
    // Pour ranger cc dans un char, il faut un cast :
    char c= (char)cc;
    ...
  }
  f.close();
}
```

## 2.5 Méthodes

Nous donnons ici une selection de méthodes utiles, en nous concentrant sur les classes `Reader` et `Writer`. Pour les classes `InputStream` et `OutputStream`, les noms des méthodes sont les mêmes, en remplaçant « `char` » par « `byte` ».

### 2.5.1 Constructeurs

#### Les classes orientées fichier

Ces classes ont toute un constructeur qui prend comme argument un *nom* de fichier, et un constructeur qui prend comme argument un objet `File` :

**`FileOutputStream`** `(File file)`
`throws` **`IOException`**

**`FileOutputStream`** `(String name)`
`throws` **`IOException`**

**`FileOutputStream`** `(String name, boolean append)`
`throws` **`FileNotFoundException`**

Ouvre le fichier `name` en écriture, en le créant si nécessaire. Si `append` est vrai, les écritures se feront à la *fin* du fichier. Si le fichier n'existe pas *et* ne peut pas être créé, alors une exception `FileNotFoundException` est levée.

**`FileInputStream`** `(File file)`
`throws` **`FileNotFoundException`**

**`FileInputStream`** `(String name)`
`throws` **`FileNotFoundException`**

Les classes `FileReader` et `FileWriter` ont les mêmes constructeurs que `FileInputStream` et `FileOutputStream`, respectivement.

#### Les classes `OutputStreamWriter` et `InputStreamReader`

Ces deux classes permettent de faire le pont entre les classes orientée octets et les classes orientées caractères. Plus précisément, étant donné un flot orienté octets, elles permettent de créer le flot orienté caractères correspondant, en encapsulant l'objet-flot d'origine dans un `Reader` ou un `Writer`.

Les constructeurs disponibles sont :

**InputStreamReader** (InputStream in)

**OutputStreamWriter** (OutputStream out)

### classes orientées chaînes

Les classes StringReader et StringWriter sont très simples :

**StringReader** (String s)
crée un StringReader qui lit dans s.

**StringWriter** ()
crée un StringWriter. La chaîne produite sera récupérable grâce à la méthode toString.

Les classes de *lecture* qui travaillent sur des tableaux et non sur des objets sont plus complexes :

**CharArrayReader** (char[] buf)
crée un lecteur qui lit dans le tableau buf.

**CharArrayReader** (char[] buf, int offset, int length)
crée un lecteur qui lit dans le tableau buf, en commençant à offset, et avec une longueur de length (soit donc entre la case offset incluse et la case offset + length exclue).

Par contre les classes d'écriture ont un constructeur très simple :

**CharArrayWriter** ()

Les classes ByteArrayOutputStream et ByteArrayInputStream ont les mêmes constructeurs que leurs équivalent en Writer et en Reader, en remplaçant les chars par des bytes.

### Autres classes

Les autres classes fonctionnent généralement comme des filtres. Elles disposent donc typiquement d'un constructeur qui prend comme argument un objet de la même famille que l'objet à créer (Writer pour Writer, Reader pour Reader, InputStream pour InputStream et OutputStream pour OutputStream). Ainsi, le constructeur de la classe LineNumberReader est :

**LineNumberReader** (Reader in)

### 2.5.2 Reader

```
int read ()
                    throws IOException
```

lit un caractère sur l'entrée, et renvoie son code, ou -1 si la fin du fichier est atteinte.

```
int read (char [] buf)
                    throws IOException
```

Remplit le tableau `buf` avec les caractères lus. Attention, cette méthode n'alloue pas le tableau.

La méthode renvoie le nombre de caractères lus, ou -1 si la fin du fichier est atteinte.

```
int read (char [] buf, int dep, int longueur)
                    throws IOException
```

Copie les caractères lus dans `buf`, en commençant à l'indice `dep`, en en lisant au maximum `longueur` caractères.

La méthode renvoie le nombre de caractères lus, ou -1 si la fin du fichier est atteinte.

```
void close ()
                    throws IOException
```

ferme le lecteur.

### 2.5.3 Writer

#### Méthodes générales

```
void write (int c)
                    throws IOException
```

Écrit le caractère dont le code est `c`. Notez que cette méthode fonctionne correctement que `c` soit un `char` ou un `int`.

```
void write (char [] s)
                    throws IOException
```

écrit le contenu de `s`.

```
void write (char[] s, int off, int longueur)
                    throws IOException
```

écrit les caractères de `s` compris entre l'indice `off` et `off + longueur`.

```
void write (String s)
                    throws IOException
```

écrit la chaîne s.

`void` **`flush`** `()`
`throws` **`IOException`**

Réalise effectivement la copie sur le support. C'est utile dans le cas de classes comme `BufferedWriter`, où les données sont stockées temporairement en mémoire.

`void` **`close`** `()`
`throws` **`IOException`**

ferme le `Writer`

**FileWriter**

La classe FileWriter permet d'écrire du texte dans un fichier. Son emploi est simple. On notera les deux contructeurs suivants :

**`FileWriter`** `(String name)`
`throws` **`IOException`**

Ouvre en écriture le fichier de nom `name`.

**`FileWriter`** `(String name, boolean append)`
`throws` **`IOException`**

Ouvre en écriture le fichier de nom `name`. Si `append` est `true`, alors l'écriture se fera à la fin du fichier, sans écraser le texte qui se trouve éventuellement au début.

## 2.6 La classe RandomAccessFile

Cette classe permet d'ouvrir un fichier dans lequel on pourra, lire, écrire, et se placer à n'importe quelle position.

### 2.6.1 Ouverture et fermeture

**`RandomAccessFile`** `(String nom, String mode)`
`throws` **`FileNotFoundException`**

Crée un RandomAccessFile ouvert sur le fichier nommé `nom`. Le `mode` peut être `r` pour un fichier ouvert en lecture seule, ou `rw` pour un fichier ouvert en lecture/écriture.

`void` **`close`** `()`
`throws` **`IOException`**

ferme le fichier

### 2.6.2 Lecture et écriture

Les méthodes de lecture permettent de lire des données *binaires*.

`int` **`read`** `()`
`throws` **`IOException`**

lit un octet non signé. On a de même que pour les `Reader`, des méthodes `read` à un et trois arguments, qui permettent de remplir des tableaux.

On dispose, de plus, des méthodes `readBoolean`, `readByte`, `readChar`, `readDouble`, `readFloat`, `readInt`, `readLong`, `readShort`, `readUnsignedByte`, `readUnsignedShort` pour lire une valeur d'un type de base. Par exemple :

`boolean` **`readBoolean`** `()`
`throws` **`IOException`**

lit un booléen.

Attention, `readByte` lit un octet *signé* (entre -127 et 128.) Les méthodes d'écriture sont similaires : on a d'une part trois méthodes `write` :

`void` **`write`** `(byte []b, int dep, int longueur)`
`throws` **`IOException`**

écrit les octets du tableau `b`, de l'indice `dep` à l'indice `dep` + 1.

et d'autre part, une méthode `write`*`XXX`* par type de base, comme par exemple `writeChar`.

Notez que *Java* définit l'ordre dans lequel sont écrit les octets des données lues, de manière indépendantes de l'architecture. Ainsi, un fichier binaire créé par java sur Macintosh sera utilisable aussi sur PC

### 2.6.3 Déplacement

`long` **`getFilePointer`** `()`
`throws` **`IOException`**

retourne la position courante.

`void` **`seek`** `(long pos)`
`throws` **`IOException`**

fixe la position courante. `pos` est le nombre de caractères depuis le début du fichier. La prochaine lecture ou la prochaine écriture auront lieu à partir de la nouvelle position.

`long` **`length`** `()`
`throws` **`IOException`**

retourne la longueur de ce fichier

`void` **`setLength`** `(long newLength)`
`throws` **`IOException`**

fixe la longueur du fichier à `newLength`.

## 2.7 Sérialisation

Java permet d'écrire ou de lire *directement* des objets dans un fichier. On utiliser ce système pour sauvegarder un objet complexe, presque sans effort.

On peut, par exemple, écrire :

```
TreeSet s;
...
// Code qui remplit s
...
ObjectOutputStream o= new ObjectOutputStream(New FileOutputStream("toto.sav"));
o.writeObject(s);
o.close();
```

Et le contenu de notre TreeSet sera sauvé dans le fichier « `toto.sav` ».

Pour relire les donnée, c'est à peine plus compliqué :

```
ObjectInputStream f= new ObjectInputStream(New FileInputStream("toto.sav"));
TreeSet s= (TreeSet)f.readObject();
f.close();
```

Remarquez le cast, dû au fait que `readObject` renvoie un `Object.`

### 2.7.1 Conditions d'emploi

Pour sauver un objet dans un ObjectOutputStream, la classe de l'objet doit implémenter l'interface (vide) `Serializable.` De plus, les champs de l'objet doivent eux aussi appartenir à des classes `Serializable`. Ainsi, comme `TreeSet` est `Serializable` (voir doc) notre exemple précédent ne fonctionnera que si les données contenues dans le `TreeSet` sont elles-mêmes `Serializable.` Nous suggérons donc de déclarer `Serializable` toute classe potentiellement utilisable avec la sérialisation.

On rappelle que l'implémentation d'interface est héritable. Si une classe est `Serializable,` toutes ses héritières le seront.

### 2.7.2 Champs non sauvegardés

Si un champ est déclaré `transient,` sa valeur n'est pas sauvée dans le fichier.

## 2.8 La classe StreamTokenizer

### 2.8.1 Introduction

La classe StreamTokenizer permet de lire de manière souple un fichier texte, en le découpant en *mots.* Ces mots sont appelés *token.* Le *Tokenizer* fonctionne en deux étapes : tout d'abord, une méthode (`nextToken()`) permet de lire le mot suivant. Ensuite, il est possible d'interroger

le Tokenizer sur la valeur de ce mot, et éventuellement sur son type (par exemple, si c'est une chaîne de caractère, une ponctuation, ou un nombre).

L'exemple qui suit présente une utilisation possible de StreamTokenizer. Il s'agit d'un petit programme qui lit sur l'entrée standard. Si on tape des nombres, ceux-ci sont ajouté à un total. En tapant « `print` », on obtient l'affichage de ce total.

La difficulté pour réaliser ce genre de programme est qu'on ne sait pas *a priori* ce qu'on va lire : `print` ou un nombre. Le programme en lui même est simple, mais, sans StreamTokenizer, sa programmation serait assez délicate. L'avantage avec StreamTokenizer est qu'on lit le token suivant sans se demander ce que c'est, et qu'on prend une décision *après* sa lecture, ce qui facilite la programmation.

Démonstration de StreamTokenizer

```
import java.io.*;

public class TestTok {

    public static void addition() throws IOException {
        double total= 0.0;
        int token;
        // Création d'un StreamTokenizer sur l'entrée standard
        StreamTokenizer s=
            new StreamTokenizer(new InputStreamReader(System.in));
        // On lit l'entrée jusqu'à ce que le texte soit fini
        while( (token= (s.nextToken())) != StreamTokenizer.TT_EOF)
            {
                //Dans token on a le type du mot lu : mot, nombre, ou autre
                switch (token) {
                case StreamTokenizer.TT_NUMBER:
                    // Si c'est un nombre, sa valeur est dans nval :
                    total+= s.nval;
                    break;
                case StreamTokenizer.TT_WORD:
                    // Si c'est un mot, elle est dans sval
                    if (s.sval.equals("print"))
                        System.out.println("total= " + total);
                    else
                        System.out.println("le mot clef " +
                                           s.sval + " est inconnu");
                    break;
                    // Pour le reste des caractères, le token
                    // a comme valeur le code du caractère
                default:
                    System.out.println("chaîne inattendue: "
                                       + (new Character((char)token)));
                }
            }
    }

    public static void main(String args[])  throws IOException {
        addition();
```

```
    }
}
```

En règle générale, le schéma de l'utilisation d'un StreamTokenizer est le suivant :

```
// Initialisation
StreamTokenizer s= .... ;
// Configuration :
  //   On appelle des fonctions diverses permettant
  //   De modifier le comportement du StreamTokenizer :
  s.lowerCaseMode(true);
  ...
  // Boucle de lecture :
  while(s.nextToken() != StreamTokenizer.TT_EOF)
    {
      case (s.ttype) {
        ...
      }
    }
```

## 2.8.2 Documentation

### Constructeurs

Il existe deux constructeurs pour les `StreamTokenizer,` mais l'un d'entre eux est obsolète. N'employez donc que le suivant :

**`StreamTokenizer`** `(Reader r)`
Construit un StreamTokenizer qui lit sur l'entrée `r`.

### Méthodes

Les méthodes peuvent être groupées en deux ensembles : celles qui permettent de *configurer* le StreamTokenizer, et celles qui sont utilisées lors de la lecture du texte.

Certaines méthodes de **configuration** permettent de fixer le comportement global d'un `StreamTokenizer`. Par exemple, `parseNumbers` lui demande d'analyser les nombres comme des nombres et non comme des suites de caractères. D'autres méthodes permettent de gérer le comportement d'un caractère en particulier.

`void` **`eolIsSignificant`** `(boolean flag)`
Si flag est `true`, alors la fin de ligne est traité comme un caractère « ordinaire ». Sinon, on la considère comme un espace. Par défaut, c'est cette seconde option qui est prise.

`void` **`lowerCaseMode`** `(boolean flag)`
Si flag est vrai, alors le texte est automatiquement traduit en minuscule, ce qui est utile pour les langages qui ignorent les différences majuscules/minuscules (ou différences de *casse* en langage typographique).

`void` **`parseNumbers`** `()`
Si cette méthode est appelée, les nombres seront considérés comme tels. Le `StreamTokenizer` fourni par défaut analyse les nombres (d'où une légitime interrogation : pourquoi diantre les concepteurs ont-ils fourni cette méthode, qui ne sert à rien, et pas de méthode `noParseNumbers,` qui aurait servi à quelque chose.

`void` **`slashSlashComments`** `(boolean flag)`
Si flag est vrai, « // » introduit un commentaire. Le texte qui suit est ignoré jusqu'à la fin de la ligne.

`void` **`slashStarComments`** `(boolean flag)`
Si flag est vrai, les commentaire de type C sont utilisés. Le texte compris entre « /* » et « */» est ignoré.

`void` **`resetSyntax`** `()`
Remet le StreamTokenizer à zéro. *Tous* les caractères sont alors considérés comme « ordinaires ». Un caractère ordinaire est traité tout seul, comme par exemple un signe de ponctuation.

Les méthodes qui suivent permettent de spécifier comment traiter des caractères donnés. Par exemple, si l'on écrit

```
tok.commentChar('#');
```

le caractère # introduira une ligne de commentaires (c'est à dire que la ligne en question sera sautée par le `StreamTokenizer`).

`void` **`commentChar`** `(int ch)`
Le caractère `ch` introduit des lignes de commentaires.

`void` **`ordinaryChar`** `(int ch)`
Le caractère `ch` est considéré comme un caractère « ordinaire ».

`void` **`ordinaryChars`** `(int low, int hi)`
Tous les caractères dont les codes sont compris entre low et hi sont considérés comme ordinaires.

`void` **`quoteChar`** `(int ch)`
Le caractère `ch` est utilisé comme guillemets.

`void` **`whitespaceChars`** `(int low, int hi)`
Tous les caractères dont les codes sont compris entre low et hi sont considérés comme des espaces.

`void` **`wordChars`** `(int low, int hi)`
Tous les caractères dont les codes sont compris entre low et hi sont considérés comme des lettres, qui forment des *mots*.

**Méthodes permettant d'utiliser le StreamTokenizer :** La méthode de loin la plus importante est bien entendu `nextToken`. On notera le comportement intéressant de `toString`.

`int` **`lineno`** `()`
: Renvoie le numéro de la ligne courante.

`int` **`nextToken`** `()`
: Lit le token suivant, et renvoie son code. Les espaces et les commentaires sont ignorés de manière transparente. Les codes possibles sont :

**TT_EOF** fin de fichier ;

**TT_EOL** fin de ligne (si `s.eolIsSignificant(true)` a été appelé ;

**TT_WORD** un « mot » a été lu. Un mot est une suite de lettre, c'est-à-dire de caractères déclarés comme `wordChars`. Par défaut, les lettres sont considérées ainsi.

**TT_NUMBER** un nombre (réel) a été lu.

**autre code :** tous autre code signifie qu'un caractère « ordinaire » (un signe de ponctuation par exemple) a été lu. Le code renvoyé est dans ce cas le code du signe lui-même.

N'oubliez pas que ces constantes doivent être précédées du nom de la classe `StreamTokenizer`.

`void` **`pushBack`** `()`
: Remet le dernier token lu sur le flot de lecture. Le prochain `nextToken()` le relira. Cette méthode n'est pas très souvent utile.

`String` **`toString`** `()`
: Renvoie le *token courant* sous forme de chaîne.

### Champs

Les champs utilisables de StreamTokenizer sont de deux types : des constantes, comme `TT_NUMBER` ou `TT_EOF`, et des champs variables, comme `nval` et `sval`. Il aurait sans douté été préférable que ces derniers fussent remplacés par des méthodes.

Les constantes sont décrites dans la documentation de `nextToken()`, ci-dessus. Les autres champs sont :

**nval** si le dernier token est de type `TT_NUMBER`, sa valeur numérique.

**sval** si le dernier token est de type `TT_WORD`, sa valeur sous forme de chaîne de caractères. Notez que si l'on veut la valeur d'un token quelconque sous forme de chaîne, on peut utiliser `toString()`.

**ttype** type du token courant. C'est la valeur renvoyée par le dernier `nextToken()`.

Dans le cas où on ne veut pas que les nombres soient analysés comme des nombres, il faut redéfinir la syntaxe de tous les caractères. Le code suivant convient :

```
StreamTokenizer tok= new StreamTokenizer(....);
tok.resetSyntax();
tok.wordChars('a', 'z');
tok.wordChars('A', 'Z');
tok.wordChars(128 + 32, 255);
tok.whitespaceChars(0, ' ');
```

Notez enfin que le `StreamTokenizer` est assez limité, et surtout orienté vers l'analyse de code informatique. En particulier, tous les caractères dont les codes sont supérieurs ou égaux à 256 sont traités comme des lettres.

Pour du texte non informatique, les problèmes sont bien plus complexes ; les bibliothèques java fournissent le package `java.text`.

# Chapitre 3

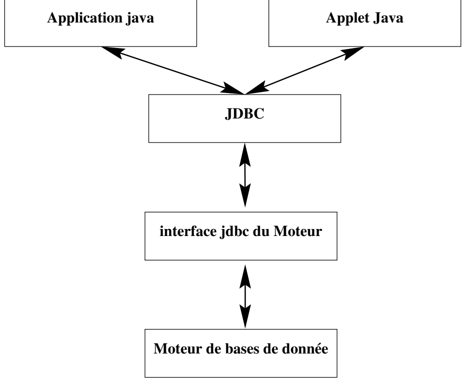
# Java et les bases de données

Jdbc : Java Data Base Connectivity.

## 3.1 Introduction à JDBC

1. Nécessité d'utiliser un langage approprié pour interroger une base de donnée (SQL) ;
2. Nécessité d'un système client serveur à cause des restrictions des *applets*.

## 3.2 Architecture




## 3.3 Un exemple : postgres

- L'interface jdbc est distribuée avec les sources, dans `postgresql-6.2/src/interfaces/jdbc` ;
- une fois compilée, la bibliothèque est utilisable sous tout ordinateur ;

– Les sources qui l'utilisent doivent contenir la ligne :

```
import java.sql.*;
```

– Pour spécifier le *driver* JDBC à utiliser, deux méthodes :

1. compiler les sources avec la ligne :
   ```
   % java -Djdbc.drivers=org.postgresql.Driver monfic.java
   ```
2. include dans le code :
   ```
   Class.forName("org.postgresql.Driver");
   ```

Attention !

Le driver doit se trouver dans le `CLASSPATH` ; s'il s'agit d'un fichier `jar`, le fichier lui même doit être dans le classpath :

```
export CLASSPATH=.:/home/titi/postgresql.jar:/usr/local/jdk1.2
```

## 3.4 établir la connexion

On utilise :

```
Connection
DriverManager.getConnection(String url,
                            String login,
                            String passwd);
```

La forme de l'URL est :

```
jdbc:sous protocole:adresse
```

### 3.4.1 Exemple :

```
try {
   Connection db;
   String url= "jdbc:postgresql://localhost/guest";
   db= DriverManager.getConnection(url, "guest", "toto");
   // manipulations diverses :
   ....
   // on a fini :
   db.close();
} catch (SQLException e) {
}
```

l'adresse est ici composée du nom du serveur postgres (`localhost`) suivi du nom de la base de donnée (ici, `guest` ; à l'IUT ce sera votre nom de login).

## 3.5 Envoyer une requête

- les **requêtes** sont représentées par la classe `Statement` ;
- les modifications de la base sont effectuées par la méthode `Statement.executeUpdate(String)` ;
- les requêtes sont effectuées par la méthode `Statement.executeQuery(String)` ;
- Le résultat d'une requête « Query » est un **ResultSet**

```
// On crée un canal de communication
Statement st= db.createStatement();
// On envoie une requête
ResultSet res= st.executeQuery("select * from Etud");
// Tant qu'il y a des lignes dans le résultat..
while (res.next()) {
  // on lit les valeurs des champs
  System.out.println("col 1 = " + rs.getString("Nom"));
}
res.close();
st.close();
```

**Notes :**

- on peut avoir plusieurs requêtes ouvertes sur la même connexion ;
- il n'est possible d'accéder à un champ qu'une fois et une seule ;
- il est nécessaire de fermer (`close`) les **Statement** et les **ResultSet**.

### 3.5.1 Méthodes

`ResultSet` **`executeQuery`** `(String requete)`
`throws` **`SQLException`**

Envoie une requête SQL, (normalement de type « `select` »), et renvoie le résultat sous forme d'un `ResultSet`. Le résultat n'est *jamais* `null`.

`int` **`executeUpdate`** `(String requete)`
`throws` **`SQLException`**

Exécute une requête de modification des données ou de la base (bref, tout ce qui n'est pas `select`). La valeur retournée normalement le nombre de lignes modifiées, ce qui a un sens pour `insert, delete, update.` Pour les autres opérateurs, le résultat est 0.

`boolean` **`execute`** `(String requete)`
`throws` **`SQLException`**

Envoie une requête SQL qui peut même envoyer plusieurs résultats. Le résultat est `true` si la première valeur renvoyée est un `ResultSet`. Nous détaillons plus avant la méthode `execute` en 3.5.3.

### 3.5.2 Méthodes applicables à un ResultSet

Les méthodes suivantes permettent d'accéder à la valeur d'une colonne, soit en passant comme argument le numéro de colonne (commençant à 1), soit le nom de la colonne : getByte getShort getInt getLong getFloat getDouble getBigDecimal getBoolean getString getBytes getDate getTime getTimestamp getAsciiStream getUnicodeStream getBinaryStream getObject

Par ailleurs, *après appel d'une de ces méthodes,* la méthode `wasNull()` permet de savoir si en fait la valeur était NULL.

### 3.5.3 Execute

La méthode `execute()` permet d'envoyer une requête, qu'elle soit de type « `select` » ou qu'elle soit une modification d'une base. Les méthodes utilisées dans l'exemple suivant permettent de récupérer des informations sur la requête. Bien entendu, dans la plupart des cas, le programmeur sait quelle est la requête, et donc utilise `executeQuery` ou `executeUpdate()`. La méthode `execute()` sera, par exemple, utilisée dans un programme où l'utilisateur pourra saisir une requête SQL quelconque.

Utilisation générale de Execute

```
stmt.execute(queryStringWithUnknownResults);
while(true) {
  int rowCount = stmt.getUpdateCount();
  if(rowCount > 0) {
    // Des données ont été modifiées
    System.out.println("Rows changed = " + count);
    stmt.getMoreResults();
    continue;
  }
  if(rowCount == 0) {
    // Modification de la Structure,
    // ou pas de changement.
    System.out.println(" Pas de ligne modifiée,
                          ou la ligne est une commande DDL");
    stmt.getMoreResults();
    continue; }
  // Si on arrive ici, il s'agit d'une requête
  ResultSet rs = stmt.getResultSet;
  if(rs != null) {
     ...
    // Il faut utiliser les métadata pour connaître
    // la liste des colonnes
    while(rs.next())
```

```
      {
        ...
        // Traiter le résultat
        stmt.getMoreResults();
        continue;
      }
    break;
    // there are no more results
  }
}
```

## 3.6 Commandes préparées

- classe `PreparedStatement` ;
- typiquement, commande utilisée plusieurs fois en changeant la valeur de certains paramètres ;
- les paramètres qui changent sont remplacés dans la commande par des « `?` » ;
- les commandes `set`*`XXX`* (où *`XXX`* est le type de la variable) permettent de spécifier la valeur des paramètres.

```
PreparedStatement pstmt =
  connec.prepareStatement("UPDATE table4 SET m = ?
                           WHERE x = ?");
...
pstmt.setString(1, "Hi");
for (int i = 0; i < 10; i++)
  {
    pstmt.setInt(2, i);
    int rowCount = pstmt.executeUpdate();
  }
```

**NULL :** pour que la valeur d'un paramètre soit `NULL` , il suffit d'utiliser la commande `setNull`

## 3.7 échappements SQL

**But :** avoir une plus grande portabilité, et faciliter la création de commandes SQL.

**Syntaxe :** Dans la chaîne de commande SQL :

```
{commande arguments}
```

**Spécifier un caractère d'échappement :**

```
stmt.executeQuery("SELECT name FROM Identifiers
        WHERE Id LIKE '\_%' {escape '\'}");
```

**Spécifier une date :**

```
{d 'yyyy-mm-dd'}
```

## 3.8 Gestion des transactions

- par défaut, chaque requête forme une transaction ;
- pour changer ce comportement, on manipule l'objet `Connection` lié à la base de donnée:

  ```
  maconnexion.setAutoCommit(false);
  ```

- ensuite :

  **`maconnexion.commit();`** valide les requêtes déjà effectuées lors de cette transaction ;

  **`maconnexion.rollback();`** annule les requêtes déjà effectuées ;

### 3.8.1 Niveau d'isolement

**But :** une transaction doit « voir » un *état* de la base. Dans le cas d'accès concurrents à la base : on peut changer le type d'accès concurrent avec la méthode de `Connection` :

```
public void {setTransactionIsolation}(int level)
                          throws SQLException
```

où `level` peut valoir :

**TRANSACTION_READ_UNCOMMITTED**

**TRANSACTION_READ_COMMITTED**

**TRANSACTION_REPEATABLE_READ**

**TRANSACTION_SERIALIZABLE**

**TRANSACTION_READ_UNCOMMITTED** on peut lire des modifications dès qu'elles sont faites. En cas de `ROLLBACK`, postérieur, les valeurs lues peuvent être fausses ;

**TRANSACTION_READ_COMMITTED** on ne peut pas lire une rangée sur laquelle il y a des modifications non validées (par commit) ;

**TRANSACTION_REPEATABLE_READ** idem ; de plus, évite le cas où la transaction lit une rangée, une autre transaction la modifie, et la première relit la rangée modifiée ; la lecture donne toujours le même résultat, d'où le nom ;

**TRANSACTION_SERIALIZABLE** le comportement est similaire à celui obtenu avec un traitement séquentiel. Empêche le cas où

1. la transaction fait un select avec une condition ;
2. une seconde transaction crée des lignes qui satisfont la condition ;
3. la première transaction refait le même select.

## 3.9 Capacités de la base de données : `DataBaseMetaData`

– Se récupère grâce à la méthode `getMetaData()` de `Connection`

– les méthodes permettent de connaître les capacités de la base. Par exemple :

`supportsSelectForUpdate()` renvoie vrai si la base permet d'utiliser un select dans un update (cf. le cours de SQL !)

des méthodes permettent d'obtenir le catalogue de la base et la liste des tables :

```
ResultSet getTables (String catalog,
   String schemaPattern, String tableNamePattern,
   String[] types)
                    throws SQLException
```

revoie un `ResultSet` décrivant les tables et les index de la base. Par exemple, pour afficher la liste des tables et index :

```
rs= meta.getTables(null , null, null, null);
while (rs.next()) {
        System.out.println(rs.getString("TABLE_NAME");
}
rs.close();
```

Les arguments de `getTables` peuvent être nuls. Les plus intéressants sont :

**types :** un tableau de chaînes de caractère, donnant le type des tables à récupérer, entre autres : `TABLE` pour les tables stricto sensu, `VIEW` pour les vues.

## 3.10 Exploration des tables

la méthode `getMetaData()` de l'interface `ResultSet` permet de récupérer le `ResultSetMetaData` associé.

### 3.10.1 méthodes de **ResultSetMetaData**

`int` **getColumnCount**`()` nombre de colonnes ;

`String` **getColumnName**`(int column)` : nom de la i$^{e}$ colonne ;

`String` **getColumnLabel**`(int column)` : titre de la colonne pour affichage ;

`int` **getColumnType**`(int column)` : type SQL de la colonne ; les valeurs possibles pour le résultat sont décrites dans `java.sql.Types`.

## 3.11 Extensions du jdbc2.0

La version 2.0 du `jdbc` propose un certain nombre d'extensions.

### 3.11.1 **ResultSet** navigables

Par défaut, on ne peut parcourir un un `ResultSet` que d'une manière : du premier au dernier élément. Le `jdbc` version 2 permet de se déplacer librement dans un `ResultSet`, et éventuellement d'en modifier les éléments. Ces options ne sont pas forcément implémentées par les drivers `jdbc.` Par exemple, le driver `postgresql` permet les déplacements, mais pas la modification.

Pour disposer de `ResultSets` modifiables, il faut le demander au moment de créer un `Statement`, en utilisant la méthode `createStatement` de la classe `Connection` :

```
Statement createStatement (int resultSetType, int
    resultSetConcurrency)
                        throws SQLException
```

**resultSetType** trois valeurs possibles :

- **ResultSet.TYPE_FORWARD_ONLY** : seul les déplacements vers l'avant sont possibles
- **ResultSet.TYPE_SCROLL_INSENSITIVE** : tout déplacement est possible. Par contre, si les données sont modifiées, et que l'on revient sur une ligne déjà visitée, la valeur visible sera la valeur d'origine et non la valeur modifiée.
- **ResultSet.TYPE_SCROLL_SENSITIVE** : tout déplacement est possible. , Si les données sont modifiées, et que l'on revient sur une ligne déjà visitée, la valeur visible sera la valeur modifiée.

**resultSetConcurrency** : règle le comportement du `ResultSet` en cas par raport aux transactions.

- **ResultSet.CONCUR_READ_ONLY** : lecture seule ;
- **ResultSet.CONCUR_UPDATABLE** : modifiable.

### Déplacement dans le ResultSet

Un `ResultSet` fonctionne comme un tableau dont les cases sont numérotées de **1** à **n**. Il dispose de plus de deux positions spéciales, `beforeFirst` et `afterLast`, aux deux extrémitées du tableau. Le curseur est à l'origine placé sur `beforeFirst`.

`void` **`last`** `()`
`throws` **`SQLException`**
se place au dernier enregistrement.

`void` **`first`** `()`
`throws` **`SQLException`**
se place au premier enregistrement.

`void` **`afterLast`** `()`
`throws` **`SQLException`**
se place après le dernier enregistrement.

`void` **`beforeFirst`** `()`
`throws` **`SQLException`**
se place avant le premier enregistrement.

`void` **`next`** `()`
`throws` **`SQLException`**
avance à l'enregistrement suivant.

`void` **`previous`** `()`
`throws` **`SQLException`**
avance à l'enregistrement précédent.

`boolean` **`absolute`** `(int i)`
`throws` **`SQLException`**
se place sur l'enregistrement numéro i. Si `i` vaut 1, c'est l'équivalent de `first`. Si `i` est négatif, on numérote à partir du *dernier* enregistrement.

La fonction renvoie `true` si le curseur pointe sur un enregistrement valide.

`boolean` **`relative`** `(int delta)`
`throws` **`SQLException`**
Déplacement relatif à la position courante. `relative(-1)` est équivalent à `previous`, et `relative(1)` à `next()`.

La fonction renvoie `true` si le curseur pointe sur un enregistrement valide.

`int` **`getRow`** `()`
`throws` **`SQLException`**
renvoie l'indice de la ligne courante.

### Modification d'un ResultSet

Un ResultSet n'est modifiable que si on l'a demandé et que le driver le gère.
Les méthodes principales sont (remplacer *XXX* par int, String...) :

```
void updateXXX (int i, XXX a)
                        throws SQLException
```
modifie le i$^e$ champ, de type *XXX*, en lui donnant la valeur `a`.

```
void updateXXX (String name, XXX a)
                        throws SQLException
```
modifie le champ nommé `name`, de type *XXX*, en lui donnant la valeur `a`.

```
void deleteRow ()
                        throws SQLException
```
détruit la ligne courante.

```
void moveToInsertRow ()
                        throws SQLException
```
se place sur une ligne spéciale, qui sert aux insertions de nouvelles données.

```
void moveToCurrentRow ()
                        throws SQLException
```
après un appel à `moveToInsertRow`, revient à sa position initiale.

```
void insertRow ()
                        throws SQLException
```
insère le contenu de la ligne d'insertion dans la base.

# Chapitre 4

# Java Server Pages et Servlets

On trouvera un répertoire d'exemples à l'adresse :

`http://www.iut.univ-paris8.fr/~rosmord/DOCS/demojsp.tgz`

Ces exemples sont conçus et testés avec tomcat4, sous `debian.` Pour pouvoir les essayer :

1. allez dans répertoire `testTomcat` ;
2. lancez le script : `./tom start` ;
3. attendez un peu ;
4. connectez-vous à l'adresse `http://localhost:9180/index.html`

## 4.1 Architecture d'une application web

- Rappel sur notions de client et serveur ;
- javascript et les *applets* java s'exécutent sur le client ;
- au contraire, les applications `cgi,` les scripts `php,` les `jsp` et les `servlets` s'exécutent sur le serveur.
- Exemple d'un script php : le client demande une page, qui est en fait un script php. Le serveur exécute le script, dont le résultat (typiquement une page html) est expédié comme réponse, puis visualisé par le client.

## 4.2 Introduction aux jsp

### 4.2.1 principes

Le principe d'une jsp est similaire à celui d'un script php. Il s'agit de texte `html` mélangé à du code, écrit en java dans le cas des jsp.

Une page `jsp` est traitée par le serveur (tomcat par exemple), qui crée le code d'une classe java associée. Par exemple, pour une page jsp nommée `test.jsp`, `tomcat` crée une classe `test$jsp.java`. À chaque page jsp correspond donc une classe java. Le code html et une partie du code jsp proprement dit sont utilisés par le serveur pour produire la méthode `_jspService`, qui est appelée à l'exécution de la *jsp* pour créer la page résultat.

### 4.2.2 Balises principales des jsp

**`<% %>`** : cette balise encadre des *instructions* java qui feront partie de la méthode `_jspService` utilisée pour produire la page résultat :

```
<html><body>
bonjour, voici la valeur de 3 + 4 :
<%    // du code java
   int x= 3 + 4;
   out.write("" + x);
%>
</body></html>
```

**`<%= %>`** : cette balise encadre une *expression* java. La valeur de cette expression sera affichée dans la page résultat :

```
<html><body>
bonjour, voici la valeur de 3 + 4 :
<%= 3 + 4 %>
</body></html>
```

**`<%! %>`** : cette balise permet d'ajouter des méthodes et des champs à la classe qui correspond à la *jsp*.

**`<%@ page ... %>`** : la balise « directive de page » sert à configurer un certain nombre d'options pour la jsp. Parmis celles-ci, citons :

**`<%@ page import="..." %>`** permet de réaliser d'importer des packages java, comme par exemple :

```
<%@ page import="java.sql.*" %>
```

**`<%@ page include="`*`toto.jsp`*`" %>`** : inclut le fichier *toto.jsp*.

**`<%@ page errorPage="`*`toto.jsp`*`" %>`** : en cas de levée d'exception sur cette page, on affichera *toto.jsp*. La variable `exception` contiendra l'objet exception correspondant.

**`<%@ page isErrorPage="true" %>`** : signifie que cette page est une page de traitement d'exception (voir directive précédente).

### 4.2.3 Les champs d'une jsp

Dans une *jsp*, les variables suivantes sont définies :

**out** : un objet de classe `JspWriter` (qui descend de `PrintWriter`). Permet d'écrire dans le résultat. Les méthodes `print` et `println` sont définies pour les principaux types de bases.

**request** : objet de type `HttpServletRequest`, qui représente la requête qui a amené sur cette page. Deux méthodes sont particulièrement utiles : `getParameter(String name)` qui renvoie la valeur d'un paramètre (un champ d'un formulaire html). Pour les champs multi-valués, on peut utiliser `getParameterValues(String name)`.

**response** :

**page**

**session** :

**application**

**config**

**pageContext**

### 4.2.4 Installation des jsp

Les *jsp* doivent se trouver dans des sous-répertoires (ou en-dessous) du répertoire `webapps` ; leur nom doit se terminer par le suffixe `jsp`.

## 4.3 Introduction aux servlets

### 4.3.1 principe des servlets

Une servlet est une classe java.

La requête (provenant généralement d'un formulaire) est traitée par la méthode `doPost` ou par la méthode `doGet` de la servlet, selon le type de requête. La servlet construira ensuite la réponse, en manipulant un objet de classe `HttpServletResponse`.

### 4.3.2 Installation des servlets

L'installation des *servlets* est plus complexe que celle des *jsp*. Il nous faut donc détailler plus précisément le répertoire `webapps`. Ce répertoire contient des sous-répertoire, chacun d'eux étant une « application » web. Chaque sous-répertoire peut contenir un nombre quelconque de

jsp et de servlets. Si vous regardez le contenu de `test1,` vous constaterez que rien n'indique *a priori* qu'il contient une servlet :

```
bash-2.05a$ pwd
/home/rosmord/Prog/testTomCat/webapps
bash-2.05a$ ls -R test1
test1:
index.html  WEB-INF

test1/WEB-INF:
classes web.xml

test1/WEB-INF/classes:
TestServlet.class

bash-2.05a$
```

Cependant, il en existe bien une, accessible à l'adresse : `http://localhost:9180/test1/salut`.

En fait, les servlets sont contenues dans le répertoire `WEB-INF`. Ce répertoire contient :

– un fichier `web.xml`, qui permet de configurer les servlets et en particulier de les associer à une `URL` ;

– un répertoire `classes`, qui contient des classes java (et éventuellement des packages) ;

– un répertoire `lib,` qui contient des archives `jar` (par exemple, les bibliothèques nécessaires pour jdbc).

Quand une servlet est écrite, il faut la compiler, et placer le résultat dans le répertoire classes. Les commandes suivantes peuvent convenir :

```
export CLASSPATH
CLASSPATH=/usr/share/java/servlet-2.3.jar
javac Hello.java
mv Hello.class testTomCat/webapps/appli1/WEB-INF/classes/
```

Il reste cependant à préciser comment on lance la servlet. C'est le travail du fichier `web.xml`. Pour chaque servlet, il contient deux informations :

– la classe associée à la servlet ;

– l'url associée à la servlet.

Ainsi, le fichier suivant permet de préciser que la servlet « bonjour » est associée à l'url « hello » et à la classe « `TestServlet` ». Évitez d'utiliser des accents ou autres caractères non ascii [1].

web.xml

```
<?xml version="1.0" ?>
<!DOCTYPE web-app
```

1. ou précisez `<?xml version="1.0" encoding="iso-8859-1"?>` en en-tête.

```
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd"
>
<!-- les formules magiques ci-dessus sont a reprendre telles quelles :-) -->
<!-- Le corps du fichier est compris dans la balise web-app : -->
  <web-app>

    <!-- la balise servlet lie la servlet
       a la classe qui l'implemente. -->

    <servlet>
      <servlet-name>bonjour</servlet-name>
      <servlet-class>monPackage.TestServlet</servlet-class>
    </servlet>

    <!-- lie la servlet a une url -->
    <servlet-mapping>
      <servlet-name>bonjour</servlet-name>
      <url-pattern>/salut</url-pattern>
    </servlet-mapping>
  </web-app>
```

### 4.3.3 Les servlets, leur vie, leur œuvre

La base du fonctionnement des servlets est que l'une des deux méthodes `doGet` et `doPost` est appelée pour traiter la requête du serveur. Il est important de comprendre que l'objet servlet n'est pas détruit après une requête : il sera réutilisé pour les consultations ultérieures de la même servlet.

À la première consultation d'une *servlet,* le serveur crée un objet de la classe considérée ; il réutilisera cet objet lors des consultations ultérieures de la servlet. Une fois l'objet créé, la méthode `init` est appelée pour l'initialiser. `init` n'est appelée qu'une seule fois dans la vie de la servlet.

Ensuite, selon la requête, la méthode `doGet` ou `doPost` est appelée. Ces deux méthodes sont très similaires, et prennent les mêmes arguments :

```
public void doGet (HttpServletRequest request,
     HttpServletResponse response)
                        throws ServletException, IOException
```

**request** représente la requête envoyée à la servlet. Utilisé pour accéder aux paramètres de celles-ci.

**response** permet de construire la réponse à la requête.

Voici un exemple simple de compteur en mémoire :

TestServlet.java

```
import java.io.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;

public class TestServlet extends HttpServlet {

    int valeur;

    public void init() throws ServletException {
        super.init();
        valeur=0;
    }

    public void doGet(HttpServletRequest httpServletRequest,
                      HttpServletResponse httpServletResponse)
                throws ServletException, IOException {
        valeur++;
        // On précise que le résultat est une page de texte simple :
        // (pour du html, ce serait : text/html)
        httpServletResponse.setContentType("text/plain");
        // out permettra d'écrire ce résultat :
        Writer out= httpServletResponse.getWriter();
        // On crée la page.
        out.write("valeur du compteur : "+ valeur);
    }

}
```

La dernière méthode importante est `destroy`, qui est appelée lors de la destruction de la servlet.

**Attention !** une servlet peut être détruite quand le serveur le juge opportun. Il ne faut donc pas compter sur la conservation des données en mémoire entre deux appels de servlets.

## 4.4 Les javabeans

Les protocoles web classiques ne gèrent pas de session. Typiquement, un utilisateur demande une page web au serveur, et celui-ci la renvoie. Il n'est pas prévu initialement de conserver sur le serveur des informations sur l'utilisateur, comme un mot de passe utilisé pour interroger une base de données, ou une « corbeille » (liste de produits commandés. Il existe plusieurs méthodes pour contourner ces difficultés. L'architecture JSP/Servlets propose une solution simple et élégante, les *javabeans.* Un javabean est un objet java, identifié par un nom (une chaîne de caractères), dont la durée de vie peut dépasser la simple requête http.

Les beans sont utilisables à partir des jsp

### 4.4.1 Les classes de beans

Pour qu'un objet java puisse être utilisé comme *bean,* les conditions sont très simples :

Sa classe doit :

- implémenter `java.io.Serializable` ;
- disposer d'un constructeur par défaut (i.e. sans argument) ;
- avoir des accesseurs (impérativement nommés `setXX` et `getXX`, où *XX* est le nom d'un champ.
- la classe *doit* être explicitement située dans un package.

La classe suivante présente un bean très simple :

Le bean Compteur

```
package test;

public class Compteur implements java.io.Serializable {
    private int valeur;

    public Compteur(int v) {
        valeur= v;
    }

    public Compteur() {
        valeur= 0;
    }

    public void incremente() {
        valeur++;
    }

    public int getValeur()  {
        return this.valeur;
    }

    public void setValeur(int argValeur) {
        this.valeur = argValeur;
    }
}
```

### 4.4.2 Accès aux beans depuis les *jsp*

La jsp qui suit montre une utilisation possible d'un bean :

testBean.jsp

```
<!doctype html PUBLIC '-//W3C//DTD HTML 3.2//EN'>
<html>
  <head>
    <title>test Beans 1</title>
  </head>
  <body>
    <!-- déclaration du bean -->
```

```
    <jsp:useBean id="cpt" scope="application" class="test.Compteur"/>

    <!-- utilisation du bean -->
    Le compteur vaut : <jsp:getProperty name="cpt" property="valeur"/>

        Quand on fixe la valeur, on utilise en plus l'attribut <tt>param</tt>.

        <%
                // Utilisation du bean par du code java :
                cpt.incremente();
        %>
  </body>
</html>
```

Les balises spécifiques au beans sont :

**<jsp:useBean ... />** permet de déclarer un bean et de lui donner un nom et une durée de vie. L'objet correspondant est créé s'il n'existe pas, et réutilisé s'il existe déjà. Les attributs à remplir sont :

**id :** Le nom donné au bean. Ce nom permet de l'identifier et de le récupérer. Par ailleurs, le bean est accessible dans le code java de la *jsp* à travers une variable qui a le nom indiqué. D'où le code

```
<%
        cpt.incremente();
%>
```

Dans notre exemple.

**scope :** durée de vie du bean. Peut être fixée à `page`, `request`, `session` ou `application`. En pratique, les deux derniers sont les plus intéressants. L'option `session` désigne une session de travail de l'utilisateur. La session commence à la connexion, et se termine quand l'utilisateur arrête son navigateur, ou après un temps d'inactivité paramétrable (fixable par la méthode `setMaxInactiveInterval(int interval)` de la classe `HttpSession` ; cette méthode prend comme argument une durée en *secondes*). Chaque utilisateur connecté aura sa propre version d'un bean session. L'option `application` signifie que le bean existe pour toute la durée de vie du serveur `tomcat` ; le bean est alors unique pour tous les utilisateurs.

**class :** la classe utilisée pour implémenter le bean. Le nom de la classe doit être de la forme `nompackage.NomClasse`.

**<jsp:getProperty ... />** cette balise sera remplacée dans la page `HTML` résultat par la valeur d'un des attributs du bean. Les attributs à remplir sont :

**name** : nom du bean (celui défini précédemment par `id="..."` ;

**property** : nom de la propriété, qui doit être un des champs définis par le bean. Pour que cela fonctionne, la classe bean doit disposer d'accesseurs correspondant au même nom.

**`<jsp:setProperty ... />`** permet de fixer la valeur d'un champ du bean. Les attributs sont :

**name** : nom du bean ;

**property** : nom du champ ;

**value** : valeur à donner au champ.

### 4.4.3 accès aux beans depuis les servlets

Il est aussi possible de créer et de lire des beans depuis des servlets. La procédure diffère selon la portée (*scope*) du bean.

#### session

On doit procéder en deux temps. Il faut d'abord récupérer un objet de classe `HttpSession` :

```
HttpSession session= request.getSession(true);
```

On peut ensuite récupérer ou donner une valeur d'un bean en utilisant :

- les méthodes `getValue` et `putValue` dans la version 3 de tomcat ;
- les méthodes `getAttribute` et `setAttribute` dans la version 4.

Seul le nom des méthodes change. Leur utilisation est la même.

`void` **`setAttribute`** `(String nomBean, Object bean)`
crée un nouveau bean :

```
session.setAttribute("cpt", new Compteur());
```

`Object` **`getAttribute`** `(String nomBean)`
récupère l'objet bean associé à un nom. Par exemple :

```
Compteur c= (Compteur)session.getAttribute("cpt");
c.incremente(); // ...
```

#### application

La méthode `getServletContext()` de la classe `GenericServlet` (dont hérite toute servlet) renvoie un objet de classe `ServletContext`, qui représente l'application. Cet objet dispose lui aussi de méthodes `setAttribute` et `getAttribute`.

## 4.5 Architecture typique d'une application jsp

Pour avoir le code le plus propre possible, on utilise servlets et jsp *uniquement* dans la couche interface utilisateur. Pour le reste de l'application, on utilise des classes java normales, typiquement séparées en plusieurs couches : logique applicative, modèle, persistence.

De plus, on utilise les *jsp* uniquement pour les affichages [2]. Le traitement des requêtes est quant à lui confié à des *servlets.*

### 4.5.1 Redirection vers une jsp

Quand une *servlet* s'est exécutée, il faut bien évidemment préciser ce qui doit s'afficher sur le navigateur du client. Il est possible qu'une servlet fasse traiter ses affichage par une jsp. Pour cela, elle utilise un objet de classe `RequestDispatcher` :

```
String url= "adresse de la jsp à afficher";
// On crée le dispatcher en précisant l'url cible.
RequestDispatcher dispatcher= getServletContext().getRequestDispatcher(url);
// on redirige la requête courante. Notez que les paramètres de
// doGet() ou doPost() sont transmis à la jsp :
dispatcher.forward(request, response);
```

Il existe aussi une méthode `sendRedirect,` d'emploi plus simple. Cependant, cette dernière est beaucoup moins efficace, car elle demande *au client* de procéder à une redirection.

Généralement, la servlet a traité la requête, et a modifié ou créé des données. La `jsp` vers laquelle on redirige le traitement aura souvent pour fonction d'afficher le résultat de ces traitements. Ces données doivent donc être passées à la `jsp.` On pourrait le faire par l'intermédiaire de javabeans, mais pour éviter de charger le serveur, la meilleure manière est de d'ajouter ces donnée dans le champ `request,` en utilisant la méthode `setAttribute` de la classe `ServletRequest:`

`void` **`setAttribute`** `(java.lang.String name, java.lang.Object o)`
: Stocke `o` sous le nom `name` dans la requête. Si `o` est `null,` alors l'attribut est supprimé dans la requête.

```
String url= "adresse de la jsp à afficher";
RequestDispatcher dispatcher= getServletContext().getRequestDispatcher(url);

dispatcher.forward(request, response);
```

## 4.6 Le suivi des sessions

La gestion des sessions s'effectue en utilisant souvent les *cookies.* Cependant, les utilisateurs ne les acceptent pas toujours. Il existe une autre manière pour gérer les numéros de session, c'est

2. Notez que rien n'interdit de confier l'affichage à des servlets. Cependant, les spécialistes de *design* web n'étant généralement pas des programmeurs, on préfèrera les *jsp* pour la présentation.

de le passer comme paramètre d'une requête à l'autre, en le codant dans les `url`. Il est possible de faire automatiquement gérer les deux modes par le serveur. Pour cela, les `url` doivent être spécifiées en utilisant la méthode `encodeURL` de la classe `HttpServletResponse`.

```
<begin>
Page <a href="<%= response.encodeURL("next.jsp") %>">suivante</a>.
<end>
```

Quand l'utilisateur accepte les cookies, le numéro de session sera stocké comme un cookie, et l'URL sera « `next.jsp` ». Quand il les refuse, le numéro de session sera inclus dans l'URL :

URL générée par encodeURL()

```
next.jsp;jsessionid=FAC06A1658A6C485294302CD336BA48F
```

## 4.7 Création de nouvelles balises pour les jsp

Pour simplifier la vie des programmeurs de page web, et leur cacher autant que possible la couche java, il est possible de créer de nouvelles balises. Le code exécuté par ces balises sera défini dans des classes java.

De telles *bibliothèqyes de balises (taglibs)* sont disponibles sur le web. Voir par exemple : `http://jakarta.apache.org/taglibs`.

On crée pour cela une *bibliothèque de tags, dont l*

### 4.7.1 Une balise simple

### 4.7.2 Une boucle

### 4.7.3 Partage de variables

## 4.8 Quelques patterns

# Chapitre 5

# Java et XML

## 5.1 Le langage XML

### 5.1.1 Introduction

– pourquoi XML : format *générique* de textes *structurés.* Les fichiers complexes en format texte (exemple : sources povray, java...) ne sont pas facile à analyser. Il faut généralement décrire le format à l'aide d'une grammaire, puis écrire un programme en conséquence. Cela demande une certaine expérience. XML est (relativement) simple ; de plus le même code peut analyser n'importe quel document XML.

– historique de xml : sgml, html. XML descend de SGML (structured generic markup language), utilisé au départ pour écrire des documentations techniques.

– domaines d'emplois de xml.

1. format de sauvegarde et d'échange portable. Les dernières versions de word utilisent XML, de même qu'Openoffice. Noter que les documents Openoffice sont sauvegardés sous la forme d'un fichier zippé contenant plusieurs documents XML. ;
2. format de documents sémantiques. Les documents XML sont lisibles et manipulables par les navigateurs web récents.
3. base de données XML.

### 5.1.2 Documents XML

**documents bien formés**

Un document xml doit impérativement respecter les règles suivantes :

– il est sauvegardé en codage UTF-8 (ou 16), sauf indication contraire. Notez qu'un document sauvé en vrai ASCII (7 bits) satisfait cette condition, puisque UTF-8 coïncide avec l'ASCII pour les codes de 0 à 127.

– il commence par une en-tête de la forme :

```
<?xml version="1.0"?>
```

– le corps du document est placé entre une balise ouvrante et une balise fermante ;

– les balises (*tags*) sont toujours groupées par deux : une balise ouvrante (`<Nom>`) et la balise fermante (`</Nom>`) correspondante.

– ces balises sont correctement parenthésées. On ne doit pas les croiser. Par exemple : `<A><B>du texte</A></B>` est *incorrect.* On devrait avoir `<A><B>du texte</B></A>`.

– Lorsque le contenu d'une balise est *toujours vide*, la balise ouvrante est aussi fermante. En *xml*, le tag `IMG` de html se coderait :

```
<img src="...."/>
```

– les valeurs des attributs sont *toujours* indiquées entre guillemets.

Quand un document est codé dans un code différent de l'UTF-8, il doit le préciser dans son en-tête. *Attention,* cette règle est vraie même si les caractères non `UTF-8` n'apparaissent que dans des commentaires. Ainsi,

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

déclare que le document est codé en iso-8859-1, le codage standard sous Unix.

Exemple de document XML

```
<?xml version="1.0" encoding="iso-8859-1"?>
<promotion id="première année info">
  <etudiant id="1">
    <nom>Turing</nom>
    <prenom>Alan</prenom>
    <passe/>
  </etudiant>
  <etudiant id="2">
    <nom>Lovelace</nom>
    <prenom>Ada</prenom>
    <passe/>
  </etudiant>
</promotion>
```

### Document Type Definition (DTD)

Une `DTD` décrit la structure que peut avoir un type de document `XML`. Il existe de nombreuses dtd déjà faites, et il est possible (et relativement aisé) d'en créer de nouvelles. Au lieu de réinventer la poudre, on aura cependant intérêt à réutiliser tout ou partie de `DTD` existantes.

La DTD suivante décrit un format (très simplifié) utilisable pour un livre. Celui-ci est constitué d'une suite non vide de chapitre, chacun de ceux-ci étant formé d'un titre, d'un contenu, et d'un commentaire optionnel.

livre.dtd

```
<!ELEMENT  livre (chapitre)+>

<!ELEMENT  chapitre     (titre,contenu,commentaire?)>

<!ELEMENT  titre        (#PCDATA)>

<!ELEMENT  contenu      (#PCDATA)>

<!ELEMENT  commentaire  (#PCDATA)>

<!ATTLIST  commentaire
             id         ID                              #IMPLIED
             type       (accord|desaccord)              "desaccord"
             auteur     CDATA                           #IMPLIED  >
```

Une DTD décrit, entre autres, les éléments qui composent un document (les balises), les attributs que ceux-ci peuvent avoir, ainsi qu'un certain nombre de

**Déclaration des éléments** La déclaration des éléments est la partie la plus complexe de la DTD. Elle a la forme :

```
<!ELEMENT nomElement (DescriptionElement)>
```

où : *nomElement* est le nom de l'élément (attention, XML est sensible à la casse du texte) ; *descriptionElement* explique ce que peut contenir l'élément en utilisant un langage spécial. Pour décrire les éléments de ce langage, prenons comme convention que tout groupe en *caractères obliques* correspond à une construction de la liste.

***nomElement*** : un autre élément ;

***A,B*** : la séquence *A*, *B* puis *C* ;

***A|B*** : *A* ou *B*. ;

***A**** : *A* répété 0 ou plusieurs fois ;

***A+*** : *A* répété 1 ou plusieurs fois ;

***A?*** : optionellement *A* ;

**#PCDATA** : du texte analysé.

Ainsi, la déclaration :

```
<!ELEMENT section (titre, auteur?, (paragraphe|image)*)>
```

indique qu'un élément section est composé d'un élément *titre,* suivi optionellement d'un élément *auteur,* suivi d'une suite éventuellement vide de paragraphes et/ou d'images.

Quand un élément peut contenir directement du texte, il y a quelques contraintes :

– `#PCDATA` doit apparaître en premier dans la description de son contenu ;

– si le `#PCDATA` n'est pas seul, la description a forcément la forme :

```
<!ELEMENT nomElement (#PCDATA|nomElt1|nomElt2|...)*>
```

par exemple, voici deux déclarations possibles :

```
<!-- un titre est simplement composé de texte -->
<!ELEMENT titre (#PCDATA)>
<!-- un paragraphe peut contenir du texte et des notes de bas de page -->
<!ELEMENT paragraphe (#PCDATA|note)*>
```

Le bout de texte suivant est valide par rapport à la définition proposée :

```
<section><titre>Le langage XML</titre>
  <paragraphe>
    Voici du texte<note>et une note</note> et encore du texte.
  </paragraphe>
  <paragraphe> Un second paragraphe</paragraphe>
  <image source="toto.jpg"/>
  <paragraphe> Un troisième paragraphe</paragraphe>
</section>
```

Dans certains cas, un élément peut ne pas avoir de contenu. On le déclare alors :

```
<!ELEMENT nomElement EMPTY>
```

Par exemple :

```
<!ELEMENT image EMPTY>
```

**Déclaration des attributs** Les attributs d'un élément sont déclarés de la manière suivante :

```
<!ATTLIST NomElement
          NomAttribut1 TypeAttribut1 DefautAttribut1
          NomAttribut2 TypeAttribut2 DefautAttribut2
          ...
>
```

où *NomElement* est le nom de l'élément dont on déclare les attributs, *NomAttribut1* est un nom d'attribut.

*«TypeAttribut1» peut avoir plusieurs valeurs différentes, notamment*

– *`CDATA` : l'attribut a comme valeur une chaîne de caractères quelconque ;*

– une liste de la forme `(A|B|...)` : A,B... sont les différentes valeurs possibles. Exemple :

```
<!ATTLIST rendezVous
    jour (lundi|mardi|mercredi|jeudi|vendredi|samedi|dimanche)
           #REQUIRED>
```

– `ID` : un identificateur. Quand on l'utilisera dans un document la valeur utilisée devra être unique.

– `IDREF` : référence à un identificateur dans le même document.

– `NMTOKEN` : un nom XML bien formé (une suite de lettres, de chiffres, de souligné (_ ), de « : »).

«DefautAttribut1» précise si la valeur de l'attribut est nécessaire ou non, et éventuellement si celui-ci a une valeur par défaut. La chaîne peut être :

– `'Valeur par défaut'` : une valeur par défaut pour l'attribut ;

– `#REQUIRED` : l'attribut est nécessaire ;

– `#IMPLIED` : l'attribut est optionnel.

**documents valides**

Un document xml est dit valide s'il est bien formé et qu'il respecte une DTD.

### 5.1.3 XSLT

**Documentation :** `http://www.w3.org/TR/1999/REC-xslt-19991116`.

Le langage XSLT (XML Stylesheet Transformation) permet de créer un document XML, HTML ou texte à partir d'un document XML d'origine. Par rapport au système CSS1 (feuilles de styles HTML), XSLT a l'avantage de pouvoir modifier la structure du document auquel il est appliqué.

Une feuille de style XSLT est un document xml. Elle contient des règles de formatage qui s'appliquent aux différentes parties d'un documents.

La feuille `XSL` ci-dessous permet de produire un document `html` avec une la table des matières.

doc.xsl

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  >

  <xsl:output method="html"/>

  <!--
```

```
    règle à appliquer au document dans son ensemble :
    on commence par générer l'en-tête, puis une table des
    matières, puis on met en forme le corps du document
    -->

  <xsl:template match="mondocument">
    <html>
      <header>
      </header>
      <body>
        <!-- la table des matières -->
        <ul>
          <xsl:for-each select="chapitre/titre">
            <li>
              <a href="#{generate-id(..)}">
              <xsl:number count="chapitre"/><xsl:text>. </xsl:text>
              <xsl:value-of select="."/>
              </a>
            </li>
          </xsl:for-each>
        </ul>
        <!-- le corps du document -->
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>

  <!-- On transforme les titres en titres html,
       en créant un identificateur pour la table des matières
    -->
  <xsl:template match="titre">
    <h1><a name="{generate-id(..)}">
      <xsl:number count="chapitre"/> <xsl:text>. </xsl:text>
      <xsl:apply-templates/>
      </a>
    </h1>
  </xsl:template>

  <!-- Les commentaires ne sont pas affichés : -->
  <!-- Sans cette règle, leur texte serait recopié tel quel. -->
  <xsl:template match="commentaire">
  </xsl:template>
</xsl:stylesheet>
```

Pour l'appliquer, on peut placer la déclaration

```
<?xml-stylesheet href="doc.xsl" type="text/xsl"?>
```

dans le document xml à mettre en forme. Les navigateurs webs récents reconnaîtrons ce type de déclaration. Java permet d'appliquer une feuille de style XSL à un document quelconque.

### 5.1.4 Schémas XML

## 5.2 Les API Java

`DOM` et `SAX` sont des spécifications d'API pour manipuler des documents XML. Selon le type de documents, et les manipulations souhaitées, l'une peut être préférable à l'autre.

### 5.2.1 Document object model (DOM)

`DOM` est basée sur le principe que le document XML est analysé pour construire une structure de données arborescente qui le représente. DOM spécifie les structures utilisées en termes de classes.

Comme une application qui utilise `DOM` construit une représentation complète du document `XML`, elle est bien adaptée quand il est nécessaire de naviguer dans ce dernier. Par contre, si le document est de grande taille, le modus operandi de `DOM` implique qu'il sera intégralement chargé.

Les étapes de l'utilisation de DOM en java sont :

1. obtenir (et configurer) un analyseur DOM ;
2. l'appliquer au document ;
3. parcourir/manipuler le document.

#### Obtention et appel d'un analyseur DOM.

Il suffit de procéder comme ci-dessous.

```
DocumentBuilderFactory factory= DocumentBuilderFactory.newInstance();
DocumentBuilder builder= factory.newDocumentBuilder();
Document doc= builder.parse("exemple1.xml");
```

La variable `doc` contient alors le document analysé.

#### Manipulation d'un document DOM.

La plupart des interfaces qui décrivent une partie d'un fichier xml analysé par DOM descendent de l'interface `org.w3c.dom.Node`. Ces interfaces définissent des méthodes en lecture et en écriture. C'est-à-dire qu'il est possible de modifier un document en mémoire (la classe `Transformer` permet ensuite éventuellement de sauver ce document).

Ces interfaces sont principalement : `Attr`, `Comment`, `Document`, `Element`, `Text`.

Tout texte balisé est représenté par un `Element`, y compris le document dans son ensemble. Cependant, celui-ci est inclus dans un `Node` de type `Document`.

*Comme toutes ces interfaces descendent de `Node`, le mieux est de considérer les méthodes de cette interface.*

*`Node`* ***`appendChild`*** *`(Node newChild)`*
: *ajoute un fils à ce nœud*

*`NamedNodeMap`* ***`getAttributes`*** *`()`*
: *récupère les attributs de ce nœud. Ne fonctionne que sur un `Element` (renvoie null pour les autres nœuds).*

*`NodeList`* ***`getChildNodes`*** *`()`*
: *récupère la liste des enfants de ce nœud.*

*`Node`* ***`getNextSibling`*** *`()`*
: *le frère de ce nœud*

*`String`* ***`getNodeName`*** *`()`*
: *le nom de ce nœud*

*`short`* ***`getNodeType`*** *`()`*
: *le type de ce nœud, parmis `ATTRIBUTE_NODE` `CDATA_SECTION_NODE` `COMMENT_NODE` `DOCUMENT_FRAGMENT_NODE` `DOCUMENT_NODE` `DOCUMENT_TYPE_NODE` `ELEMENT_NODE` `ENTITY_NODE` `ENTITY_REFERENCE_NODE` `NOTATION_NODE` `PROCESSING_INSTRUCTION_NODE` `TEXT_NODE` (à faire précéder de `Node.`).*

*`String`* ***`getNodeValue`*** *`()`*
: *la valeur de ce nœud*

*`Node`* ***`getParentNode`*** *`()`*
: *le père de ce nœud*

*`boolean`* ***`hasAttributes`*** *`()`*
: *retourne vrai si ce nœud a des attributs*

*`boolean`* ***`hasChildNodes`*** *`()`*
: *retourne vrai si ce nœud a des enfants*

*`Node`* ***`insertBefore`*** *`(Node newChild, Node refChild)`*
: *insère `newChild` avant `refChild`*

*`Node`* ***`removeChild`*** *`(Node oldChild)`*
: *supprime `oldChild`*

*`void`* ***`setNodeValue`*** *`(String nodeValue)`*
: *The value of this node, depending on its type; see the table above.*

### Méthodes des classes NodeList et NamedNodeMap

*int* ***getLength*** *()*
: *récupère le nombre de nœuds dans la liste.*

*Node* ***item*** *(int i)*
: *récupère le i$^{e}$ nœuds dans la liste*

### Méthodes de la classe Document

*Element* ***getDocumentElement*** *()*
: *récupère l'élément racine du document.*

### Méthodes de la classe Element

*String* ***getAttribute*** *(String name)*
: *renvoie la valeur de l'attribut name*

*Attr* ***getAttributeNode*** *(String name)*
: *renvoie l'attribut name*

*String* ***getTagName*** *()*
: *Nom de l'élément.*

### Méthodes de la classe Attr

*String* ***getName*** *()*
: *renvoie le nom de l'attribut*

*String* ***getValue*** *()*
: *renvoie la valeur de l'attribut*

### Méthodes de la classe Text

*String* ***getData*** *()*
: *récupère le texte associé à cet élément.*

### Exemple

TestDOM.java

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.Text;
```

```
// Lit un document et l'affiche de manière indentée.
public class TestDOM {
        // Utilisé pour indenter l'affichage.
        int profondeur= 0;

        private void dumpNode(Node node) {
                switch (node.getNodeType()) {
                        case Node.ELEMENT_NODE :
                                dumpElement((Element) node);
                                break;
                        case Node.TEXT_NODE :
                                dumpText((Text) node);
                                break;
                        default :
                                }
        }

        private void printSpaces() {
                for (int i= 0; i < profondeur; i++)
                        System.out.print(" ");
        }

        private void print(String s) {
                System.out.print(s);
        }

        private void dumpElement(Element elt) {
                printSpaces(); print(elt.getNodeName()+ ":");
                NodeList l= elt.getChildNodes();
                profondeur++;
                for(int i= 0; i < l.getLength(); i++) {
                        dumpNode(l.item(i));
                }
                profondeur--;
        }

        private void dumpText(Text txt) {
                printSpaces();
                print(txt.getData());
        }

        public void dumpDocument(Document doc) {
                dumpElement(doc.getDocumentElement());
        }

        public static void main(String[] args) throws Exception {
                TestDOM nav= new TestDOM();
                DocumentBuilderFactory factory=
                    DocumentBuilderFactory.newInstance();
                DocumentBuilder builder= factory.newDocumentBuilder();
```

```
                Document doc= builder.parse("exemple1.xml");
                nav.dumpDocument(doc);
        }
}
```

## 5.2.2 Simple API for Xml Parsing (SAX)

SAX fonctionne selon les principes du design pattern « builder ». Une méthode est appelée Pour chaque élément du document XML. SAX ne construit aucune structure ; il est donc bien adapté à des parcours simples d'un document. Par contre, la construction de structures évoluées est assez complexe (et réclame typiquement l'utilisation d'une pile).

Les étapes de l'utilisation de SAX en java sont :

1. création d'un analyseur SAX ;
2. création d'un builder pour le document (en étendant la classe `DefaultHandler`[1].) ;
3. appel de l'analyseur sur le document.

### Création d'un analyseur SAX

La création de l'analyseur se fait aussi en deux étapes :

```
// Construction d'un créateur d'analyseur
SAXParserFactory factory= SAXParserFactory.newInstance();
// Configuration. Ici on demande un analyseur non validant.
factory.setValidating(false);
// création de l'analyseur
SAXParser parser= factory.newSAXParser();
```

### Création d'un builder pour le document

On doit écrire une classe dont les méthodes seront appelées pour traiter les divers éléments du document `XML`. Cette classe étend généralement la classe `DefaultHandler`, qui fournit des implémentation vide des méthodes. On ne redéfinit normalement qu'une partie des méthodes.

`void` **`characters`** `(char[] ch, int start, int length)`
: appelée pour traiter du texte. Attention, le texte à traiter commence dans le tableau ch à la position `start`, et a pour longueur `length`.

`void` **`endDocument`** `()`
: appelé à la fin du document.

1. plus précisément `org.xml.sax.helpers.DefaultHandler`

*void* ***endElement*** *(String namespaceURI, String localName, String qName)*
*appelé à la fin d'un élément. Le nom de l'élément est stocké dans* `localName` *ou* `qName`*, voire les deux, selon la configuration du système. La distinction entre ces deux noms est liée à l'utilisation possible d'« espaces de nomages », qui sont l'équivalent* `XML` *des packages* `java`*.*

*void* ***processingInstruction*** *(String target, String data)*
*appelé pour traiter les instructions entre* `<? ... ?>`

*void* ***startDocument*** *()*
*appelée au début du document.*

*void* ***startElement*** *(String namespaceURI, String localName, String qName, Attributes atts)*
*appelée au début d'un élément. Le nom de l'élément est traité soit dans* `localName`*, soit dans* `qName`*. Les attributs sont stockés dans le tableau associatif* `atts`

**traitement des attributs :** *les attributs sont gérés au travers d'une interface spécifique,* `Attributes`*. Les méthodes les plus importantes de cette interface sont :*

*int* ***getLength*** *()*
*renvoie le nombre d'attributs.*

*String* ***getQName*** *(int index)*
*renvoie le nom du index$^{e}$ attribut.*

*String* ***getValue*** *(int idx)*
*renvoie la valeur du i$^{e}$ attribut.*

*String* ***getValue*** *(String qName)*
*renvoie la valeur d'un attribut dont on connaît le nom.*

**Utilisation de l'espace de nomage** *Le comportement de l'analyseur est gouverné par deux propriétés :* `http://xml.org/sax/features/namespaces` *et* `http://xml.org/sax/features/namespace-prefixes`*. Leur valueur peut être consultée et fixée à l'aide des méthodes :*

*boolean* ***getFeature*** *(String name)*
*throws* ***SAXNotRecognizedException, SAXNotSupportedException***

```
void setFeature (String name, boolean value)
                        throws SAXNotRecognizedException,
     SAXNotSupportedException
```

Si `http://xml.org/sax/features/namespace-prefixes` est à vrai, la valeur `qName` sera fixée. Elle contient le nom complet de la balise, préfixée par le nom de l'espace de nomage.

Inversement, `http://xml.org/sax/features/namespace-prefixes` indique que `namespaceURI` et `localName` seront renseignés.

### Appel de l'analyseur sur le document

On analyse le document en appelant la méthode `parse` de la classe `SAXParser`. Exemple :

```
// Création du builder (étend DefaultHandler)
TestXML handler= new TestXML();
// Création du constructeur d'analyseurs :
SAXParserFactory factory= SAXParserFactory.newInstance();
// On veut dans cet exemple un analyseur non validant
factory.setValidating(false);
// construction de l'analyseur.
SAXParser parser= factory.newSAXParser();
// On construit une InputSource à partir du nom du fichier à traiter
InputSource src= new InputSource("toto.xml");
// On lance l'analyse
// premier argument : document à analyser
// second argument : builder à utiliser.
parser.parse(src, handler);
```

### Utilisation de SAX (ou DOM) sans DTD

Il arrive parfois qu'un document `XML` fasse référence à une DTD dont on ne dispose pas. Dans ce cas, les bibliothèques java signaleront un problème même si on ne demande pas de validation. En effet, la DTD n'est pas utilisée uniquement pour valider le document, mais aussi pour fixer les valeurs de certains attributs (valeurs par défauts), remplacer les entités par leurs valeurs... Quand on ne dispose pas de la bonne DTD, la méthode à suivre est d'en fournir une qui soit vide. On peut redéfinir pour cela la méthode `resolveEntity`, qui sera appelée lors de l'analyse pour récupérer le code qui décrit une entité donnée (ici la dtd). Supposons que nous voulions lire un document utilisant la dtd `file:///office.dtd`, sans disposer de celle-ci. On pourra écrire la méthode `resolveEntity` suivante :

```
        public InputSource resolveEntity(String publicId, String systemId)
                throws SAXException {
                if (systemId.equals("file:///office.dtd")) {
                        // On construit à la volée une dtd vide
                        return new InputSource(
```

```
                    new StringReader(
                     "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"));
          } else
                  // Renvoyer null => le système  normal  de
                  // résolution de noms sera utilisé.
                  return null;
      }
```

## 5.3 XSL et java

On peut utiliser des feuilles de style `XSLT` à partir de java. Un point très intéressant est que la source et la destination de ces transformations peuvent être, non seulement de simples fichiers, mais aussi des documents `DOM` ou des analyseurs `SAX`.

TestXSL.java

```
import javax.xml.transform.*;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

public class TestXSL {
  public static void main(String[] args) {
    try {
      // Objet qui représente la feuille de style xslt
      Source xslSource= new StreamSource("doc.xsl");
      // Document d'origine
      Source xmlSource= new StreamSource("exemple.xml");
      // Document html créé par le filtre
      Result result= new StreamResult("result.html");
      // Construction du filtre XSLT :
      // - on récupère un créateur de filtre
      TransformerFactory tfactory= TransformerFactory.newInstance();
      // - on l'utilise pour créer le filtre proprement dit
      Transformer t= tfactory.newTransformer(xslSource);
      // Application du filtre. C'est fini.
      t.transform(xmlSource, result);
    } catch (TransformerException e) {
      e.printStackTrace();
    }
  }
}
```

## 5.4 Quelques outils java utilisant XML

### 5.4.1 SVG et Batik

## 5.5 Appendice : un fragment de la description du langage XML

Les documents officiels qui décrivent le langage XML ne sont pas forcément très facile à lire. Ils utilisent la notation Extended Backus-Naur Form (EBNF). Cette dernière emploie les opérateurs « | », « * », « + » et « ? » avec le même sens que les DTD. La grande différence est que toute chaîne de caractère constante est mise entre guillemets simples.

Nous allons analyser en détail la description de la déclaration d'un élément dans une `DTD`.

```
elementdecl  ::= '<!ELEMENT' Name contentspec '>'
contentspec  ::= 'EMPTY' | 'ANY' | Mixed | children

children  ::= (choice | seq) ('?' | '*' | '+')?
cp  ::= (Name | choice | seq) ('?' | '*' | '+')?
choice  ::= '('  cp (  '|'  cp )*  ')'
seq  ::= '('  cp (  ','  cp )*  ')'

Mixed  ::= '(' '#PCDATA' ( '|' Name)* ')*' | '(' '#PCDATA' ')'
```

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

# Chapitre 6

# Swing Avancé

## 6.1 Patterns en swing : l'exemple de `JTable`

### 6.1.1 Le pattern Observateur

Toutes les classes graphiques de swing sont contruites sur le schéma vue/modèle. Dans la plupart des cas, un modèle par défaut est automatiquement construit :

```
// Crée une table de 1000 lignes et 10 colonnes :
JTable table= new JTable(1000,10);
getContentPane().add(new JScrollPane(table));
```

La classe JTable permet d'accéder à ce modèle, grâce à la méthode `getModel` :

```
TableModel model = table.getModel();
model.setValueAt("Une chaîne", 3, 4);
model.setValueAt(new ImageIcon("icon.jpg"),2,2);
model.setValueAt(new Boolean(false), 1,1);
```

Le code suivant, par exemple, aura pour effet de dupliquer en temps réel les données de la table :

```
table= new JTable(1000,10);
getContentPane().add(new JScrollPane(table));
getContentPane().add(new JScrollPane(new JTable(table.getModel())));
```

L'interface `JTableModel` définit un certain nombre de méthodes, dont la classe AbstractTableModel fournit une implémentation par défaut. En pratique, quand on a besoin d'un modèle spécifique, on étend `AbstractTableModel` si c'est possible, plutôt que d'implémenter entièrement `JTableModel`.

`Class` **`getColumnClass`** `(int columnIndex)`
: Retourne la classe correspondant aux entrées de la colonne. Dans `AbstractTableModel`, la valeur retournée est toujours `Object.class`.

*int* ***getColumnCount*** *()*
: *retourne le nombre de colonnes.*

*String* ***getColumnName*** *(int columnIndex)*
: *retourne le nom de la colonne* i.

*int* ***getRowCount*** *()*
: *retourne le nombre de lignes.*

*Object* ***getValueAt*** *(int rowIndex, int columnIndex)*
: *retourne la valeur de la cellule* `rowIndex, columnIndex`.

*boolean* ***isCellEditable*** *(int rowIndex, int columnIndex)*
: *retourne vrai si la case* `rowIndex, columnIndex` *est éditable.*

*void* ***setValueAt*** *(Object aValue, int rowIndex, int columnIndex)*
: *fixe la valeur de la case* `rowIndex, columnIndex`.

*void* ***addTableModelListener*** *(TableModelListener l)*
: *ajoute un observateur, qui sera prévenu quand le modèle sera modifié.*

*void* ***removeTableModelListener*** *(TableModelListener l)*
: *supprime un observateur.*

La classe `AbstractTableModel` propose de plus des méthodes pour prévenir les observateurs :

*void* ***fireTableCellUpdated*** *(int ligne, int colonne)*
: *avertit les observateurs que la cellule* `ligne, colonne` *a été modifiée.*

*void* ***fireTableDataChanged*** *()*
: *avertit les observateurs que les données de la table ont changé.*

*void* ***fireTableRowsDeleted*** *(int firstRow, int lastRow)*
: *avertit les observateurs que les rangées d'indices compris entre i0 et i1, inclus, ont été supprimées.*

*void* ***fireTableRowsInserted*** *(int firstRow, int lastRow)*
: *avertit les observateurs que les rangées d'indices compris entre i0 et i1, inclus, ont été insérées*

*void* ***fireTableRowsUpdated*** *(int i0, int i1)*
: *avertit les observateurs que les rangées d'indices compris entre i0 et i1, inclus, ont été modifiées.*

*void* ***fireTableStructureChanged*** *()*
: *avertit les observateurs que la structure de la table a changé.*

*void* ***fireTableChanged*** *(TableModelEvent e)*
: *Expédie l'événement* `e` *qui représente les modifications faites à la table à tous ses observateurs. Généralement, on utilisera plutôt les méthodes précédentes.*

## 6.2 Le pattern *commande* en swing : actions et edits

*Le design pattern* commande *est utilisé à plusieurs reprises dans* swing, *à des fins différentes.*

### 6.2.1 Actions et le pattern *commande*

*Il arrive souvent dans une interface graphique qu'une opération soit réalisable de plusieurs façon : par un menu, un bouton, un raccourci clavier...*

### 6.2.2 Undo, classes textes et design pattern commande