Développons en Java

Jean Michel DOUDOUX

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Développons en Java

Version 0.60 béta

du 23 décembre 2002

par Jean Michel DOUDOUX

A propos de ce document

L'idée de départ de ce document était de prendre des notes relatives à mes premiers essais en Java. Ces notes ont tellement grossies que j'ai décidé de les formaliser un peu plus et de les diffuser sur internet.

Ce didactitiel est composé de cinq grandes parties :

1. les bases du langage java
2. les interfaces graphiques
3. les API avancées
4. le développement serveur
5. les outils de développement

Chacune de ces parties est composées de plusieurs chapitres.

Je souhaiterais le développer pour qu'il couvre un maximum de sujets autour du développement en Java. Ce souhait est ambitieux car l'API de Java est très riche et ne cesse de s'enrichir au fil des versions.

Je suis ouvert à toutes réactions ou suggestions concernant ce document notamment le signalement des erreurs, les points à éclaircir, les sujets à ajouter, etc. ... N'hésitez pas à me contacter : jean-michel.doudoux@wanadoo.fr

Ce document est disponible aux formats HTML et PDF à l'adresse suivante : http://perso.wanadoo.fr/jm.doudoux/java/

Ce manuel est fourni en l'état, sans aucune garantie. L'auteur ne peut être tenu pour responsable des éventuels dommages causés par l'utilisation des informations fournies dans ce document.

La version pdf de ce document est réalisée grâce à l'outils HTMLDOC 1.8.14 de la société Easy Software Products. Cet excellent outil freeware peut être téléchargé à l'adresse : http://www.easysw.com

Note de licence

Copyright (C) 1999−2001 DOUDOUX Jean Michel

Vous pouvez copier, redistribuer et/ou modifier ce document selon les termes de la Licence de Documentation Libre GNU, Version 1.1 ou toute autre version ultérieure publiée par la Free Software Foundation; les Sections Invariantes étant constitués des chapitres :

- Développons en Java, présentation,
- les techniques de base de la programmation Java,
- la syntaxe et les éléments de base de Java,
- la programmation orientée objet,
- la bibliothèque de classes Java,
- les fonctions mathématiques,
- la gestion des exceptions,
- le multitâche,
- le graphisme,
- les éléments d'interface graphique de l'AWT,
- la création d'interface graphique avec AWT,
- l'interception des actions de l'utilisateur,
- le développement d'interface graphique avec SWING,
- les applets,
- les collections,
- les flux,
- la sérialisation,
- l'interaction avec le réseau,
- l'accès aux bases de données : JDBC,
- la gestion dynamique des objets et l'instrospection,
- l'appel de méthode distantes : RMI,
- l'internationalisation,
- les composants java beans,
- le logging,
- J2EE,
- les servlets,
- les JSP (Java Server Pages),
- JSTL (Java server page Standard Tag Library),
- les frameworks,
- Java et XML,
- JNDI (Java Naming and Directory Interface),
- JMS (Java Message Service),
- Java Mail,
- JDO (Java Data Object),
- les Entreprise Java Bean,
- les services web,
- les outils du J.D.K.,
- les outils libres et commerciaux,
- Javadoc,
- java et UML,
- des normes de développement,
- les motifs de conception,
- Ant
- et les annexes

aucun Texte de Première de Couverture, et aucun Texte de Quatrième de Couverture. Une copie de la licence est incluse dans la section GNU FreeDocumentation Licence.

La version la plus récente de cette licence est disponible à l'adresse : GNU Free Documentation Licence.

Marques déposées

Java est une marque déposée de Sun Microsystems Inc.

Historique des versions

Version	Date	Evolutions
0.10 béta	15/01/2001	brouillon : 1ere version diffusée sur le web.
0.20 béta	11/03/2001	ajout des chapitres JSP et serialization, des informations sur le JDK et son installation, corrections diverses.
0.30 béta	10/05/2001	ajout des chapitres flux, beans et outils du JDK, corrections diverses.
0.40 béta	10/11/2001	réorganisation des chapitres et remise en page du document au format HTML (1 page par chapitre) pour faciliter la maintenance ajout des chapitres : collections, XML, JMS, début des chapitres Swing et EJB séparation du chapitre AWT en trois chapitres.
0.50 béta	31/04/2002	séparation du document en trois parties ajout de chapitres : logging, JNDI, java mail, services web, outils du JDK, outils lires et commerciaux, java et UML, motifs de conception compléments ajoutés aux chapitres : JDBC, javadoc, intéraction avec le réseau, java et xml, bibliothèques de classes
0.60 béta	23/12/2002	ajout des chapitres : JSTL, JDO, Ant, les frameworks ajout des sections : java et MySQL, les classes internes, les expressions régulières, dom4j compléments ajoutés aux chapitres : JNDI, design patterns, J2EE, EJB

Partie 1 : les bases du langage Java

Cette première partie est chargée de présenter les bases du langage java.

Elle comporte les chapitres suivants :

- présentation de java : introduit le langage java en présentant les différentes éditions et versions du JDK, les caractéristiques du langage et décrit l'installation du JDK
- les techniques de base de programmation : présente rapidement comment compiler et executer une application
- la syntaxe et les éléments de bases de java : explore les éléments du langage d'un point de vue syntaxique
- la programmation orientée objet : explore comment java d'utiliser son orientation objet
- la bibliothèque de classes java : propose un présentation rapide des principales API fournies avec le JDK
- les fonctions mathématiques : indique comment utiliser les fonctions mathématiques
- la gestion des exceptions : traite de la faculté de java de traiter les anomalies qui surviennent lors de l'éxécution du code
- le multitâche :

1. Présentation

Java est un langage de programmation à usage général, évolué et orienté objet dont la syntaxe est proche du C. Il existe 2 types de programmes en Java : les applets et les applications. Une application autonome (stand alone program) est une application qui s'exécute sous le contrôle direct du système d'exploitation. Une applet est une application qui est chargée par un navigateur et qui est exécutée sous le contrôle de celui ci.

Ce chapitre contient plusieurs sections :

1.1. Les différentes éditions de Java

Sun fourni gratuitement un ensemble d'outils et d'API pour permettre le développement de programmes avec Java. Ce kit, nommé JDK, est librement téléchargeable sur le site web de Sun http://java.sun.com ou par FTP ftp://java.sun.com/pub/

Le JRE contient uniquement l'environnement d'exécution de programmes Java. Le JDK contient lui même le JRE. Le JRE seul doit être installé sur les machines ou des applications java doivent être exécutées.

Depuis sa version 1.2, Java a été renommé Java 2. Les numéros de versions 1.2 et 2 désignent donc la même version.

Le JDK a été renommé J2SDK (Java 2 Software development Kit) mais la dénomination JDK reste encore largement utilisée.

le JRE a été renommé J2RE (Java 2 Runtime Edition).

Trois éditions du JDK 1.2 existent :

- J2ME : Java 2 Micro Edition
- J2SE : Java 2 Standard Edition
- J2EE : Java 2 Entreprise Edition

Sun fourni le JDK 1.2 sous les plate−formes Windows, Solaris et Linux.

La version 1.3 de Java est désignée sous le nom Java 2 version 1.3.

La documentation au format HTML des API de java est fournie séparément. Malgré sa taille imposante, cette documentation est indispensable pour obtenir des informations complètes sur les classes fournies. Le tableau ci dessous résume la taille des différents composants selon leur version pour la plateforme Windows.

	Version 1.0	Version 1.1	Version 1.2	Version 1.3	Version 1.4 béta
JDK compressé		8,6 Mo	20 Mo	30 Mo	47 Mo
JDK installé				53 Mo	59 Mo
JRE compressé			12 Mo	7 Mo	
JRE installé				35 Mo	40 Mo
Documentation compressée			16 Mo	21 Mo	
Documentation décompressée			83 Mo	106 Mo	

1.1.1. Le JDK 1.0

Cette première version est lancée officiellement en mai 1995.

1.1.2. Le JDK 1.1

Cette version du JDK est annoncée officiellement en mars 1997. Elle apporte de nombreuses améliorations et d'importantes fonctionnalités nouvelles dont :

- les java beans
- les fichiers JAR
- RMI pour les objets distribués
- la sérialisation
- l'introspection
- JDBC pour l'accès aux données
- les classes internes
- l'internationalisation
- un nouveau modèle de sécurité permettant notamment de signer les applets
- JNI pour l'appele de méthodes natives
- ...

1.1.3. Le JDK 1.2

Cette version du JDK est lancée fin 1998. Elle apporte de nombreuses améliorations et d'importantes fonctionnalités nouvelles dont :

- un nouveau modèle de sécurité basé sur les policy
- les JFC sont incluses dans le JDK (Swing, Java2D, accessibility, drag & drop ...)
- JDBC 2.0
- les collections
- support de CORBA
- un compilateur JIT est inclus dans le JDK
- de nouveaux format audio sont supportés
- ...

Java 2 se décline en 3 éditions différentes qui regroupent des APIs par domaine d'applications :

- Java 2 Micro Edition (J2ME) : contient le nécessaire pour développer des applications capable de fonctionner dans des environnements limités tels que les assistants personnels (PDA), les téléphones portables ou les systèmes de navigation embarqués

- Java 2 Standard Edition (J2SE) : contient le nécessaire pour développer des applications et des applets. Cette édition reprend le JDK 1.0 et 1.1.
- Java 2 Enterprise Edition (J2EE) : contient en plus un ensemble de plusieurs API permettant le développement d'applications destinées aux entreprises tel que JDBC pour l'accès aux bases de données, EJB pour développer des composants orientés métiers, Servlet / JSP pour générer des pages HTML dynamiques …

Le but de ces trois éditions est de proposer une solution à base java quelque soit le type de développement à réaliser.

1.1.4. Le JDK 1.3

Cette version du JDK apporte de nombreuses améliorations notamment sur les performances et des fonctionnalités nouvelles dont :

- JNDI est inclus dans le JDK
- hotspot est inclus dans la JVM
- ...

La rapidité d'exécution a été grandement améliorée dans cette version.

1.1.5. Le JDK 1.4 béta

Cette version du JDK apporte de nombreuses améliorations notamment sur les performances et des fonctionnalités nouvelles dont :

- JAXP est inclus dans le JDK pour le support de XML
- JDBC version 3.0
- new I/O API pour compléter la gestion des entrée/sortie
- logging API pour la gestion des logs applicatives
- l'outils java WebStart
- ...

Cette version devrait être proposée en version finale au début de l'année 2002.

1.1.6. Le résumé des différentes versions

Au fur et à mesure des nouvelles versions du J.D.K., le nombre de packages et de classes s'accroît :

	JDK 1.0	JDK 1.1	JDK 1.2	JDK1.3	JDK 1.4 béta
Nombre de de packages	8	23		76	
Nombre de classes	201	503		1840	

1.1.7. Les extensions du JDK

Sun fourni un certains nombres d'API supplémentaires qui ne sont fournies en standard dans le JDK.

Extension	Description
JNDI	Java Naming and directory interface Cet API permet d'unifier l'accès à des ressources. Elle est intégré au JDK 1.3

Java mail	Cet API permet de gérer des emails
Java 3D	Cet API permet de mettre en oeuvre des graphismes en 3 dimensions
Java Media	Cet API permet d'utiliser des composants multimédia
Java Servlets	Cet API permet de créer des servlets (composants serveurs)
Java Help	Cet API permet de créer des aide en ligne pour les applications
Jini	Cet API permet d'utiliser java avec des appareils qui ne sont pas des ordinateurs
JAXP	Cet API permet le parsing et le traitement de document XML. Elle est intégré au JDK 1.4

1.2. Les caractéristiques

Java est interprété	le source est compilé en pseudo code puis exécuté par un interpréteur Java (la Java Virtual Machine (JVM))
Java est indépendant de toute plate−forme	il n'y a pas de compilation spécifique pour chaque plate forme. Le code reste indépendant de la machine sur laquelle il s'exécute. Il est possible d'exécuter des programmes Java sur tous les environnements qui possèdent une Java Virtual Machine. Cette indépendance est assurée au niveau du code source grâce à Unicode et au niveau du byte code.
Java est orienté objet.	
Java est simple	le choix de ses auteurs a été d'abandonner des éléments mal compris ou mal exploités des autres langages tels que la notion de pointeurs (pas d'incident en manipulant directement la mémoire), de l'héritage multiple et de la surcharge des opérateurs
Java est fortement typé	toutes les variables sont typées et il n'existe pas de conversion automatique qui risquerait une perte de données.
Java assure la gestion de la mémoire	l'allocation de la mémoire pour un objet est automatique à sa création et Java récupère automatiquement la mémoire inutilisée grâce au garbage collector qui restitue les zones de mémoire laissées libres suite à la destruction des objets.
Java est sûr	la sécurité fait partie intégrante du système d'exécution et du compilateur. Un programme Java planté ne menace pas le système d'exploitation. Il ne peut y avoir d'accès direct à la mémoire. L'accès au disque dur est réglementé dans une applet. Les programmes fonctionnant sur le Web sous soumis aux restrictions suivantes dans la version 1.0 de Java : • aucun programme ne peut ouvrir, lire, écrire ou effacer un fichier sur le système de l'utilisateur • aucun programme ne peut lancer un autre programme sur le système de l'utilisateur • toute fenêtre créée par le programme est clairement identifiée comme fenêtre Java, ce qui interdit par exemple la création d'une fausse fenêtre demandant un mot de passe • Les programmes ne peuvent pas se connecter à d'autres sites Web que celui dont il provient.
Java est économe	

	le pseudo code a une taille relativement petite car les bibliothèques de classes requises ne sont liées qu'à l'exécution.
Java est multitâche	il permet l'utilisation de threads qui sont des unités d'exécution isolées. La JVM elle même utilise plusieurs threads.

Les programmes Java exécutés localement sont des applications, ceux qui tournent sur des pages Web sont des applets.

Les différences entre une applet et une application sont :

- les applets n'ont pas de bloc main() : la méthode main() est appelée par la machine virtuelle pour exécuter une application.
- les applets ne peuvent pas être testées avec l'interpréteur mais doivent être intégrées à une page HTML, elle même visualisée avec un navigateur sachant gérer les applets Java, ou testées avec l'applet viewer.

1.3. Les différences entre Java et JavaScript

Il ne faut pas confondre Java et JavaScript. JavaScript est un langage développé par Netscape Communications.

La syntaxe des deux langages est proche car elles dérivent toutes les deux du C++.

Il existe de nombreuses differences entre les deux langages :

	Java	Javascript
Auteur	Développé par Sun Microsystems	Développé par Netscape Communications
Format	Compilé sous forme de byte−code	Interprété
Stockage	Applet téléchargé comme un élément de la page web	Source inséré dans la page web
Utilisation	Utilisable pour développer tous les types d'applications	Utilisable uniquement pour "dynamiser" les pages web
Execution	Executé dans la JVM du navigateur	Executé par le navigateur
POO	Orienté objets	Manipule des objets mais ne permet pas d'en définir
Typage	Fortement typé	Pas de contrôle de type
Compléxité du code	Code relativement complexe	Code simple

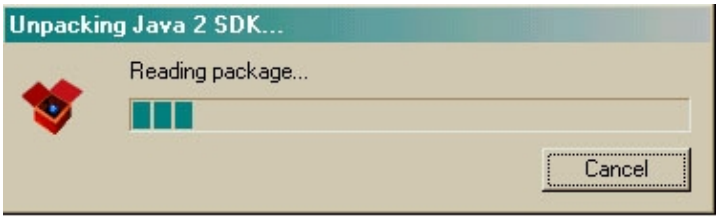
1.4. L'installation du JDK

Le JDK et la documentation sont librement téléchargeable sur le site web de Sun : http://java.sun.com

1.4.1. L'installation de la version 1.3 DU JDK de Sun sous Windows 9x

Pour installer le JDK 1.3 sous Windows 9x, il suffit d'exécuter le programme : j2sdk1_3_0−win.exe

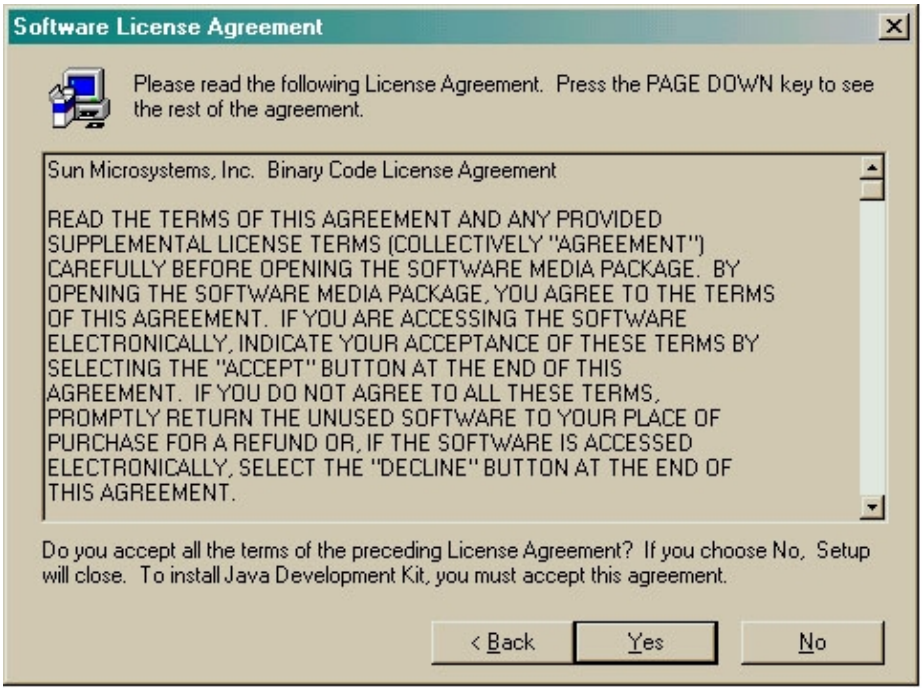
Le programme commence par désarchiver les composants.



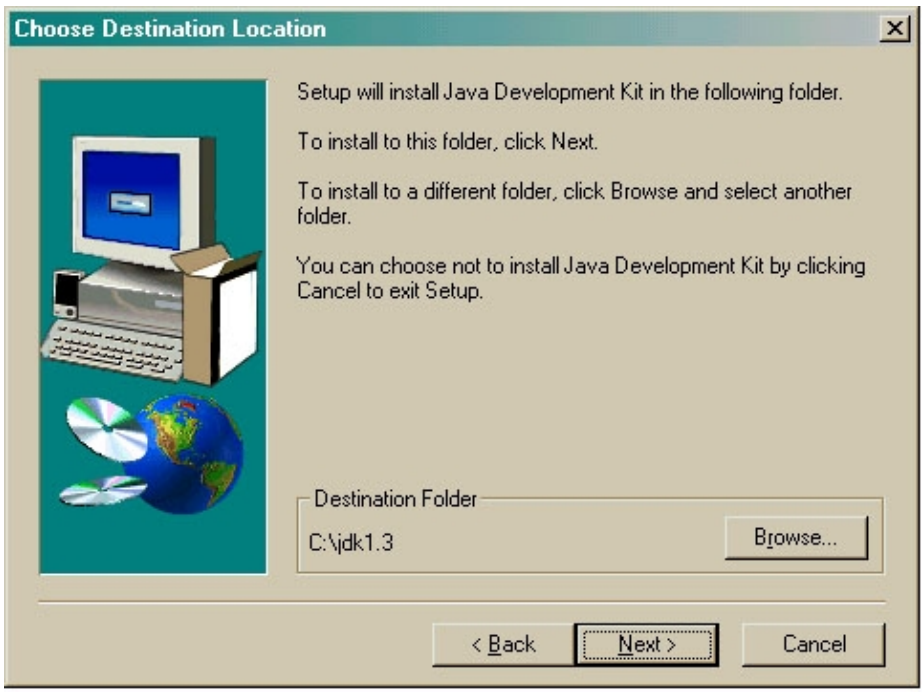
Le programme utilise InstallShield pour réaliser l'installation

L'installation vous souhaite la bienvenue et vous donne quelques informations d'usage.

L'installation vous demande ensuite de lire et d'approuver les termes de la licence d'utilisation.

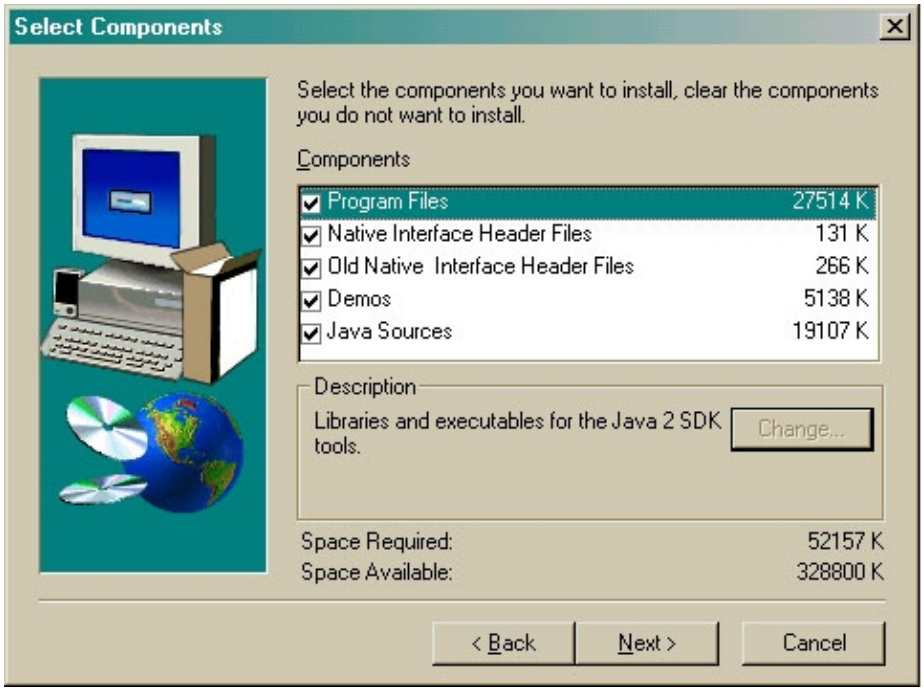


L'installation vous demande le répertoire dans lequel le JDK va être installé. Le répertoire proposé par défaut est pertinent car il est simple.



L'installation vous demande les composants à installer :

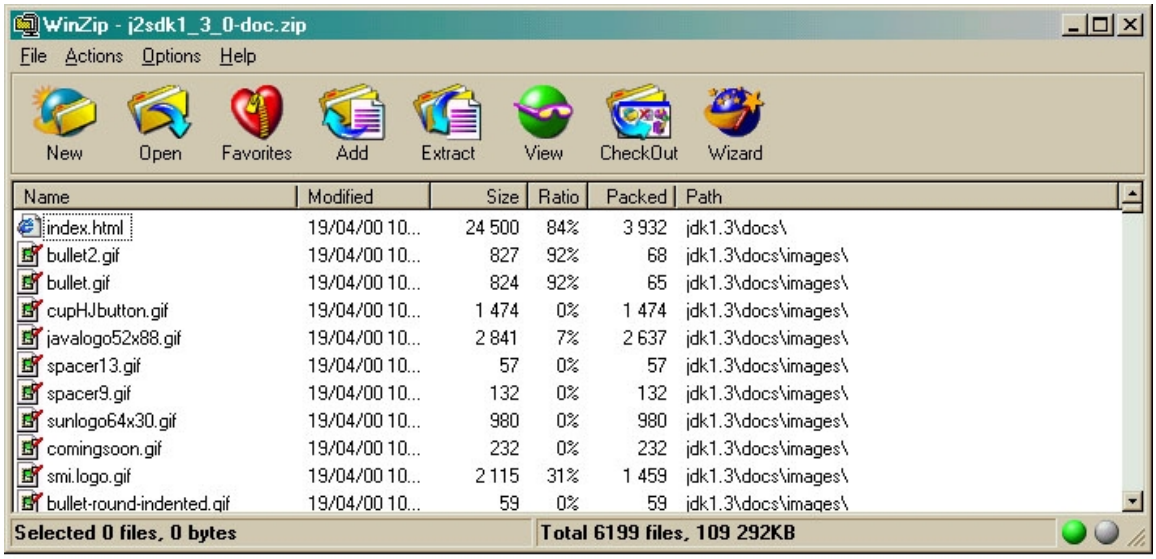
- Program Files est obligatoire pour une première installation
- Les interfaces natives ne sont utiles que pour réaliser des appels de code natif dans les programmes java
- Les démos sont utiles car ils fournissent quelques exemples
- les sources contiennent les sources de la plupart des classes java écrite en java. Attention à l'espace disque nécessaire à cet élément



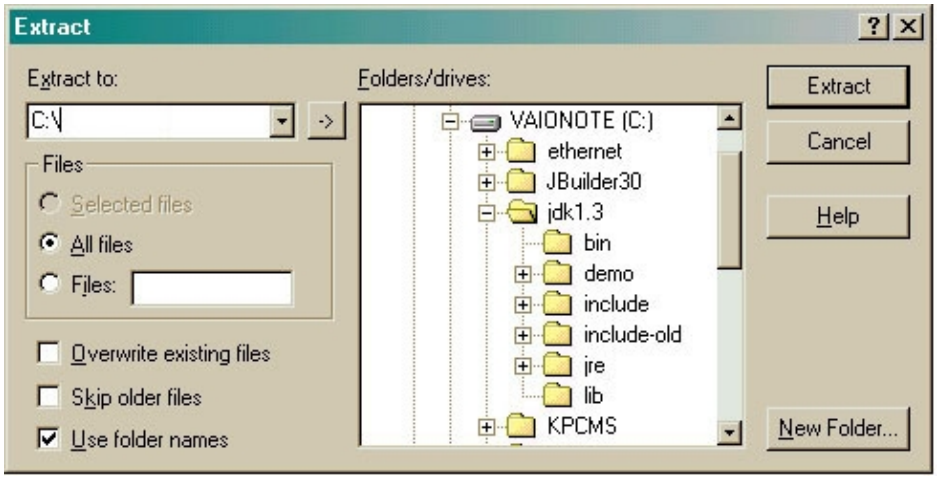
L'installation se poursuit par la copie des fichiers et la configuration du JRE

1.4.2. L'installation de la documentation sous Windows

L'archive contient la documentation sous forme d'arborescence dont la racine est jdk1.3\docs.



Si le répertoire par défaut a été utilisé lors de l'installation, il suffit de décompresser l'archive à la racine du disque C:\.



Il peut être pratique de désarchiver le fichier dans dans un sous répertoire, ce permet de reunir plusieurs versions de la documentation.

1.4.3. La configuration des variables système sous Windows

Pour un bon fonctionnement du JDK, il faut paramétrer correctement deux variables systèmes : la variable PATH qui définie les chemins de recherche des exécutables et la variable CLASSPATH qui définies les chemins de rechercher des classes java.

Pour configurer la variable PATH, il suffit d'ajouter à la fin du fichier autoexec.bat :

Exemple :
`SET PATH=%PATH%;C:\JDK1.3\BIN`

Attention : si une version antérieure du JDK était déjà présente, la variable PATH doit déjà contient un chemin vers les utilitaires du JDK. Il faut alors modifier ce chemin sinon c'est l'ancienne version qui sera utilisée. Pour vérifier la version du JDK utilisé, il suffit de saisir la commande java –version dans une fenêtre DOS.

La variable CLASSPATH est aussi définie dans le fichier autoexec.bat. Il suffit d'ajouter ligne ou de modifier la ligne existante définissant cette variable.

Exemple :
`SET CLASSPATH=.;`

Il est intéressant d'ajouter le . qui désigne le répertoire courant dans le CLASSPATH.

Il faudra ajouter par la suite les chemins d'accès aux différent packages requis par les développements.

Pour que ces modifications prennent effet dans le système, il faut redemarrer Windows ou executer ces deux instructions sur une ligne de commande dos.

1.4.4. Les éléments du JDK sous Windows

Le répertoire dans lequel a été installé le JDK contient plusieurs répertoires. Les répertoires données ci après sont ceux utilisés en ayant gardé le répertoire défaut lors de l'installation.

Répertoire	Contenu
C:\jdk1.3	Le répertoire d'installation contient deux fichiers intéressants : le fichier readme.html qui fourni quelques information et des liens web et le fichier src.jar qui contient le source java de nombreuses classes. Ce dernier fichier n'est présent que si l'option correspondante a été cochée lors de l'installation.
C:\jdk1.3\bin	Ce répertoire contient les exécutables : le compilateur javac, l'interpréteur java, le débuggeur jdb et d'une façon générale tous les outils du JDK.
C:\jdk1.3\demo	Ce répertoire n'est présent que si l'option nécessaire a été cochée lors de l'installation. Il contient des applications et des applets avec leur code source.
C:\jdk1.3\docs	Ce répertoire n'est présent que si la documentation a été décompressée.
C:\jdk1.3\include et C:\jdk1.3\include−old	Ces répertoires ne sont présent que si les options nécessaires ont été cochées lors de l'installation. Il contient des fichiers d'en−tête C (fichier avec l'extension .H) qui permettent d'inclure du code C dans le source java.
C:\jdk1.3\jre	Ce répertoire contient le JRE : il regroupe le nécessaire à l'exécution des applications notamment le fichier rt.jar qui regroupe les API. Depuis la version 1.3, le JRE contient deux machines virtuelles : la JVM classique et la JVM utilisant la technologie Hot spot. Cette dernière est bien plus rapide et celle qui est utilisée par défaut. Les éléments qui composent le JRE sont séparés dans les répertoires bin et lib selon leur nature.
C:\jdk1.3\lib	Ce répertoire ne contient plus que quelques bibliothèques notamment le fichier tools.jar. Avec le JDK 1.1 ce répertoire contenait le fichier de la bibliothèque standard. Ce fichier est maintenant dans le répertoire JRE.

2. Les techniques de base de programmation en Java

N'importe quel éditeur de texte peut être utilisé pour éditer un fichier source Java.

Il est nécessaire de compiler le source pour le transformer en J−code ou byte−code Java qui sera lui exécuté par la machine virtuelle.

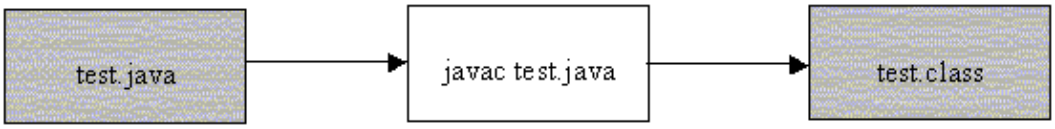
Il est préférable de définir une classe par fichier. Le nom de la classe publique et le fichier qui la contient doivent être identiques.

Pour être compilé, le programme doit être enregistré au format de caractères Unicode : une conversion automatique est faite par le JDK si nécessaire.

Ce chapitre contient plusieurs sections :

- La compilation d'un code source :
  Cette section présente la compilation d'un fichier source.
- L'exécution d'un programme et d'une applet :
  Cette section présente l'éxécution d'un programme et d'une applet.

2.1. La compilation d'un code source



Pour compiler un fichier source il suffit d'invoquer la commande javac avec le nom du fichier source avec son extension .java

```
javac NomFichier.java
```

Le nom du fichier doit correspondre au nom de la classe principale en respectant la casse même si le système d'exploitation n'y est pas sensible

Suite à la compilation, le pseudo code Java est enregistré sous le nom NomFichier.class

2.2. L'exécution d'un programme et d'une applet

2.2.1. L'exécution d'un programme

Une classe ne peut être éxécutée que si elle contient une méthode main() correctement définie.

Pour exécuter un fichier contenant du byte−code il suffit d'invoquer la commande java avec le nom du fichier source sans son extension .class

```
java NomFichier
```

2.2.2. L'exécution d'une applet

Il suffit de créer une page HTML pouvant être très simple :

Exemple

```
<HTML>
<TITLE> test applet Java </TITLE>
<BODY>
<APPLET code=« NomFichier.class » width=270 height=200>
</APPLET>
</BODY>
</HTML>
```

Il faut ensuite visualiser la page créée dans l'appletviewer ou dans un navigateur 32 bits compatible avec la version de Java dans laquelle l'applet est écrite.

3. La syntaxe et les éléments de bases de java

Ce chapitre ce compose de plusieurs sections :

- Les règles de bases
  Cette section présente les règles syntaxiques de base de java.
- Les identificateurs
  Cette section présente les règles de composition des identificateurs.
- Les commentaires
  Cette section présente les différentes formes de commentaires de java.
- La déclaration et l'utilisation de variables
  Cette section présente la déclaration des variables, les types élémentaires, les formats des type élémentaires, l'initialisation des variables, l'affectation et les comparaisons.
- Les opérations arithmétiques
  Cette section présente les opérateurs arithmétique sur les entiers et les flottants et les opérateurs d'incrémentation et de décrémentation.
- La priorité des opérateurs
  Cette section présente la priorité des opérateurs.
- Les structures de contrôles
  Cette section présente les instructions permettant la réalisation de boucles, de branchements conditionnels et de débranchements.
- Les tableaux
  Cette section présente la déclaration, l'initialisation explicite et le parcours d'un tableau
- Les conversions de types
  Cette section présente la conversion de types élémentaires.
- La manipulation des chaines de caractères
  Cette section présente la définition et la manipulation de chaine de caractères (addition, comparaison, changement de la casse ...).

3.1. Les règles de base

Java est sensible à la casse.

Les blocs de code sont encadrés par des accolades. Chaque instruction se termine par un ‘;'.

Une instruction peut tenir sur plusieurs lignes

Exemple :

```
 char

code

=

'D';
```

L'indentation est ignorée du compilateur mais elle permet une meilleure compréhension du code par le programmeur.

3.2. Les identificateurs

Chaque objet, classe, programme ou variable est associer à un nom : l'identificateur qui peut se composer de tous les caractères alphanumériques et des caractères _ et $. Le premier caractère doit être une lettre, le caractère de soulignement ou le signe dollars.

Rappel : Java est sensible à la casse.

3.3. Les commentaires

Ils ne sont pas pris en compte par le compilateur donc ils ne sont pas inclus dans le pseudo code. Ils ne se terminent pas par un ;.

Il existe trois type de commentaire en Java :

Type de commentaires	Exemple
commentaire abrégé	`// commentaire sur une seule ligne` `int N=1; // déclaration du compteur`
commentaire multiligne	`/* commentaires ligne 1` `commentaires ligne 2 */`
commentaire de documentation automatique	`/** commentaire */`

3.4. La déclaration et l'utilisation de variables

3.4.1. La déclaration de variables

Une variable possède un nom, un type et une valeur. La déclaration d'une variable doit donc contenir deux choses : un nom et le type de données qu'elle peut contenir. Une variable est utilisable dans le bloc ou elle est définie.

La déclaration d'une variable permet de réserver la mémoire pour en stocker la valeur.

Le type d'une variable peut être un type élémentaire ou un objet :

type_élémentaire variable;

classe variable ;

Exemple :

```
long nombre;
```

Rappel : les noms de variables en Java peuvent commencer par un lettre, par le caractère de soulignement ou par le signe dollars. Le reste du nom peut comporter des lettres ou des nombres mais jamais d'espaces.

Il est possible de définir plusieurs variables de même type en séparant chacune d'elles par une virgule.

Exemple :

```
int jour, mois, annee ;
```

Java est un langage à typage rigoureux qui ne possède pas de transtypage automatique lorsque ce transtypage risque de conduire à une perte d'information.

Pour les objets, il est nécessaire en plus de la déclaration de la variable de créer un objet avant de pouvoir l'utiliser. Il faut réserver de la mémoire pour la création d'un objet (remarque : un tableau est un objet en java) avec l'instruction **new**. La libération de la mémoire se fait automatiquement grâce au garbage collector.

Exemple :

```
MaClasse instance; // déclaration de l'objet

instance = new maClasse(); // création de l'objet

OU MaClasse instance = new MaClasse(); // déclaration et création de l'objet
```

Exemple :

```
int[] nombre = new int[10];
```

Il est possible en une seule instruction de faire la déclaration et l'affectation d'une valeur à une variable ou plusieurs variables.

Exemple :

```
int i=3 , j=4 ;
```

3.4.2. Les types élémentaires

Les types élémentaires ont une taille identique quelque soit la plate−forme d'exécution : c'est un des éléments qui permet à java d'être indépendant de la plate−forme sur lequel le code s'exécute.

Type	Désignation	Longueur	Valeurs	Commentaires
boolean	valeur logique : true ou false	8 bits	True ou false	pas de conversion possible vers un autre type
byte	octet signé	8 bits	−128 à 127	
short	entier court signé	16 bits	−32768 à 32767	
char	caractère Unicode	16 bits	\u0000 à \uFFFF	entouré de cotes simples dans un programme Java
int	entier signé	32 bits	−2147483648 à 2147483647	
float	virgule flottante simple précision (IEEE754)	32 bits	1.401e−045 à 3.40282e+038	
double	virgule flottante double précision (IEEE754)	64 bits	2.22507e−308 à 1.79769e+308	
long	entier long	64 bits	−9223372036854775808 à 9223372036854775807	

Les types élémentaires commencent tous par une minuscule.

3.4.3. Le format des types élémentaires

Le format des nombres entiers :

Les types byte, short, int et long peuvent être codés en décimal, hexadécimal ou octal. Pour un nombre hexadécimal, il suffit de préfixer sa valeur par 0x. Pour un nombre octal, le nombre doit commencer par un zéro. Le suffixe l ou L permet de spécifier que c'est un entier long.

Le format des nombres décimaux :

Les types float et double stockent des nombres flottants : pour être reconnus comme tel ils doivent posséder soit un point, un exposant ou l'un des suffixes f, F, d, D. Il est possible de préciser des nombres qui n'ont pas le partie entière ou décimale.

Exemple :

```
float pi = 3.141f;
double v = 3d
float f = +.1f , d = 1e10f;
```

Par défaut un littéral est de type double : pour définir un float il faut le suffixer par la lettre f ou F.

Exemple :

```
double w = 1.1;
```

Attention : `float pi = 3.141; // erreur à la compilation`

Le format des caractères :

Un caractère est codé sur 16 bis car il est conforme à la norme Unicode. Il doit être entouré par des apostrophes. Une valeur de type char peut être considérée comme un entier non négatif de 0 à 65535. Cependant la conversion implicite par affectation n'est pas possible.

Exemple :

```
/* test sur les caractères */
class test1 {
    public static void main (String args[]) {
        char code = 'D';
        int index = code - 'A';
        System.out.println("index = " + index);
    }
}
```

3.4.4. L'initialisation des variables

Exemple :

```
int nombre; // déclaration
nombre = 100; //initialisation
OU int nombre = 100; //déclaration et initialisation
```

En java, toute variable appartenant à un objet (définie comme étant un attribut de l'objet) est initialisée avec une valeur par défaut en accord avec son type au moment de la creation. Cette initialisation ne s'applique pas aux variables locales

des méthodes de la classe.

Les valeurs par défaut lors de l'initialisation automatique des variables d'instances sont :

Type	Valeur par défaut
boolean	false
byte, short, int, long	0
float, double	0.0
char	\u000

Remarque : Dans une applet, il est préférable de faire les déclarations et initialisation dans la méthode init().

3.4.5. L'affectation

le signe = est l'opérateur d'affectation et s'utilise avec une expression de la forme variable = expression. L'opération d'affectation est associatif de droite à gauche : il renvoie la valeur affectée ce qui permet d'écrire :

```
x = y = z = 0;
```

Il existe des opérateurs qui permettent de simplifier l'écritures d'une opération d'affectation associée à un opérateur mathématique :

Opérateur	Exemple	Signification
=	a=10	équivalent à : a = 10
+=	A+=10	équivalent à : a = a + 10
−=	a−=	équivalent à : a = a − 10
*=	A*=	équivalent à : a = a * 10
/=	a/=10	équivalent à : a = a / 10
%=	A%=10	reste de la division
^=	a^=10	équivalent à : a = a ^ 10
<<=	A<<=10	équivalent à : a = a << 10 a est complété par des zéros à droite
>>=	a>>=10	équivalent à : a = a >> 10 a est complété par des zéros à gauche
>>>=	a>>>=10	équivalent à : a = a >>> 10 décalage à gauche non signé

Attention : Lors d'une opération sur des opérandes de type différents, le compilateur détermine le type du résultat en prenant le type le plus précis des opérandes. Par exemple, une multiplication d'une variable de type float avec une variable de type double donne un résultat de type double. Lors d'une opération entre un opérande entier et un flottant, le résultat est du type de l'opérande flottant.

3.4.6. Les comparaisons

Java propose des opérateurs pour toutes les comparaisons :

Opérateur	Exemple	Signification

>	a > 10	strictement supérieur
<	a < 10	strictement inférieur
>=	a >= 10	supérieur ou égal
<=	a <= 10	inférieur ou égal
==	a == 10	Egalité
!=	a != 10	diffèrent de
&	a & b	ET binaire
^	a ^ b	OU exclusif binaire
|	a | b	OU binaire
&&	a && b	ET logique (pour expressions booléennes) : l'évaluation de l'expression cesse dès qu'elle devient fausse
||	a || b	OU logique (pour expressions booléennes) : l'évaluation de l'expression cesse dès qu'elle devient vraie
? :	a ? b : c	opérateur conditionnel : renvoie la valeur b ou c selon l'évaluation de l'expression a (si a alors b sinon c) : b et c doivent retourner le même type

Les opérateurs sont exécutés dans l'ordre suivant à l'intérieure d'une expression qui est analysée de gauche à droite:

- incréments et décréments
- multiplication, division et reste de division (modulo)
- addition et soustraction
- comparaison
- le signe = d'affectation d'une valeur à une variable

L'usage des parenthèse permet de modifier cet ordre de priorité.

3.5. Les opérations arithmétiques

Les opérateurs arithmétiques se notent + (addition), − (soustraction), * (multiplication), / (division) et % (reste de la division). Ils peuvent se combiner à l'opérateur d'affectation

Exemple :

```
nombre += 10;
```

3.5.1. L'arithmétique entière

Pour les types numériques entiers, Java met en oeuvre une sorte de mécanisme de conversion implicite vers le type int appelée promotion entière. Ce mécanisme fait partie des règles mise en place pour renforcer la sécurité du code.

Exemple :

```
short x= 5 , y = 15;
x = x + y ; //erreur à la compilation

Incompatible type for =. Explicit cast needed to convert int to short.
x = x + y ; //erreur à la compilation
```

```
^
1 error
```

Les opérandes et le résultat de l'opération sont convertis en type int. Le résultat est affecté dans un type short : il y a donc risque de perte d'informations et donc erreur à la compilation est émise. Cette promotion évite un débordement de capacité sans que le programmeur soit pleinement conscient du risque : il est nécessaire, pour régler le problème, d'utiliser une conversion explicite ou cast

Exemple :

```
x = (short) ( x + y );
```

Il est nécessaire de mettre l'opération entre parenthèse pour que ce soit son résultat qui soit converti car le cast a une priorité plus forte que les opérateurs arithmétiques.

La division par zéro pour les types entiers lève l'exception ArithmeticException

Exemple :

```
/* test sur la division par zero de nombres entiers */
class test3 {
    public static void main (String args[]) {
        int valeur=10;
        double résultat = valeur / 0;
        System.out.println("index = " + résultat);
    }
}
```

3.5.2. L'arithmétique en virgule flottante

Avec des valeurs float ou double, la division par zéro ne produit pas d'exception mais le résultat est indiqué par une valeur spéciale qui peut prendre trois états :

- indéfini : Float.NaN ou Double.NaN (not a number)
- indéfini positif : Float.POSITIVE_INFINITY ou Double.POSITIVE_INFINITY, + ∞
- indéfini négatif : Float.NEGATIVE_INFINITY ou Double.NEGATIVE_INFINITY, + ∞

Conformément à la norme IEEE754, ces valeurs spéciales représentent le résultat d'une expression invalide NaN, une valeur supérieure au plafond du type pour infini positif ou négatif.

X	Y	X / Y	X % Y
valeur finie	0	+ ∞	NaN
valeur finie	+/− ∞	0	x
0	0	NaN	NaN
+/− ∞	valeur finie	+/− ∞	NaN
+/− ∞	+/− ∞	NaN	NaN

Exemple :

```
/* test sur la division par zero de nombres flottants */

class test2 {
```

```
    public static void main (String args[]) {
        float valeur=10f;
        double résultat = valeur / 0;
        System.out.println("index = " + résultat);
    }
}
```

3.5.3. L'incrémentation et la décrémentation

Les opérateurs d'incrémentation et de décrémentation sont : `n++ ++n n-- --n`

Si l'opérateur est placé avant la variable (préfixé), la modification de la valeur est immédiate sinon la modification n'a lieu qu'à l'issu de l'exécution de la ligne d'instruction (postfixé)

L'opérateur ++ renvoie la valeur avant incrémentation s'il est postfixé, après incrémentation s'il est préfixé.

Exemple :

```
System.out.println(x++); // est équivalent à
System.out.println(x); x = x + 1;

System.out.println(++x); // est équivalent à
x = x + 1; System.out.println(x);
```

Exemple :

```
/* test sur les incrementations prefixees et postfixees */

class test4 {
    public static void main (String args[]) {
        int n1=0;
        int n2=0;
        System.out.println("n1 = " + n1 + " n2 = " + n2);
        n1=n2++;
        System.out.println("n1 = " + n1 + " n2 = " + n2);
        n1=++n2;
        System.out.println("n1 = " + n1 + " n2 = " + n2);
        n1=n1++;        //attention
        System.out.println("n1 = " + n1 + " n2 = " + n2);
    }
}
```

Résultat :

```
int n1=0;

int n2=0; // n1=0 n2=0

n1=n2++; // n1=0 n2=1

n1=++n2; // n1=2 n2=2

n1=n1++; // attention : n1 ne change pas de valeur
```

3.6. La priorité des opérateurs

Java définit les priorités dans les opérateurs comme suit (du plus prioriotaire au moins prioritaire)

les parenthèses	( )

les opérateurs d'incrémentation	++ −−
les opérateurs de multiplication, division, et modulo	* / %
les opérateurs d'addition et soustraction	+ −
les opérateurs de décalage	<< >>
les opérateurs de comparaison	< > <= >=
les opérateurs d'égalité	== !=
l'opérateur OU exclusif	^
l'opérateur ET	&
l'opérateur OU	|
l'opérateur ET logique	&&
l'opérateur OU logique	||
les opérateurs d'assignement	= += −=

Les parenthèses ayant une forte priorité, l'ordre d'interprétation des opérateurs peut être modifié par des parenthèses.

3.7. Les structures de contrôles

3.7.1. Les boucles

```
while ( boolean )
{
    ... // code a éxécuter dans la boucle
}
```

Le code est exécuté tant que le booléen est vrai. Si avant l'instruction while, le booléen est faux, alors le code de la boucle ne sera jamais exécuté

Ne pas mettre de ; après la condition sinon le corps de la boucle ne sera jamais exécuté

```
do {
   ...
} while ( boolean )
```

Cette boucle est au moins exécuté une fois quelque soit la valeur du booléen;

```
for ( initialisation; condition; modification) {
    ...
}
```

Exemple :
`for (i = 0 ; i < 10; i++ ) { ....}` `for (int i = 0 ; i < 10; i++ ) { ....}` `for ( ; ; ) { ... } // boucle infinie`

L'initialisation, la condition et la modification de l'index sont optionels.

Dans l'initialisation, on peut déclarer une variable qui servira d'index et qui sera dans ce cas locale à la boucle.

Il est possible d'inclure plusieurs traitements dans l'initialisation et la modification de la boucle : chacun des traitements doit etre séparé par une virgule.

Exemple :
`for (i = 0 , j = 0 ; i * j < 1000;i++ , j+= 2) { ....}`

La condition peut ne pas porter sur l'index de la boucle :

Exemple :
`boolean trouve = false;` `for (int i = 0 ; !trouve ; i++ ) {` `    if ( tableau[i] == 1 )` `    trouve = true;` `    ... //gestion de la fin du parcours du tableau` `}`

Il est possible de nommer une boucle pour permettre de l'interrompre même si cela est peu recommendé :

Exemple :
`int compteur = 0;` `boucle:` `while (compteur < 100) {` `    for(int compte = 0 ; compte < 10 ; compte ++) {` `        compteur += compte;` `        System.out.println("compteur = "+compteur);` `        if (compteur> 40) break boucle;` `    }` `}`

3.7.2. Les branchements conditionnels

```
if (boolean) {
    ...
} else if (boolean) {
    ...
} else {
    ...
```

```
}
```

```
swith (expression) {
    case constante1 :
        instr11;
        instr12;
        break;

    case constante2 :
        ...
    default :
        ...
}
```

On ne peut utiliser switch qu'avec des types primitifs d'une taille maximum de 32 bits (byte, short, int, char).

Si une instruction case ne contient pas de break alors les traitements associés au case suivant sont éxécutés.

Il est possible d'imbriquer des switch

L'opérateur ternaire : (condition) **?** valeur−vrai **:** valeur−faux

Exemple :

```
if (niveau == 5) // equivalent à total = (niveau ==5) ? 10 : 5;
total = 10;
else total = 5 ;
System.out.println((sexe == « H ») ? « Mr » : « Mme »);
```

3.7.3. Les débranchements

break : permet de quitter immédiatement une boucle ou un branchement. Utilisable dans tous les controles de flot

continue : s'utilise dans une boucle pour passer directement à l'itération suivante

break et continue peuvent s'excuter avec des blocs nommés. Il est possible de préciser une étiquette pour indiquer le point de retour lors de la fin du traitement déclenché par le break.

Une étiquette est un nom suivi d'un deux points qui définit le début d'une instruction.

3.8. Les tableaux

Ils sont dérivés de la classe Object : il faut utiliser des méthodes pour y accéder dont font parti des messages de Object tel que equals() ou getClass(). Le premier élément possède l'indice 0.

3.8.1. La déclaration des tableaux

Java permet de placer les crochets après ou avant le nom du tableau dans la déclaration.

Exemple :

```
int tableau[] = new int[50]; // déclaration et allocation

OU int[] tableau = new int[50];

OU int tab[]; // déclaration
tab = new int[50]; //allocation
```

Java ne supporte pas directement les tableaux à plusieurs dimensions : il faut déclarer un tableau de tableau.

Exemple :

```
float tableau[][] = new float[10][10];
```

La taille des tableaux de la seconde dimension peut ne pas être identiques pour chaque occurrences.

Exemple :

```
int dim1[][] = new int[3][];
dim1[0] = new int[4];
dim1[1] = new int[9];
dim1[2] = new int[2];
```

Chaque élément du tableau est initialisé selon son type par l'instruction new : 0 pour les numériques, '\0' pour les caractères, false pour les booléens et nil pour les chaines de caractères et les autres objets.

3.8.2. L'initialisation explicite d'un tableau

Exemple :

```
int tableau[5] = {10,20,30,40,50};
int tableau[3][2] = {{5,1},{6,2},{7,3}};
```

La taille du tableau n'est pas obligatoire si le tableau est initialisé à sa création.

Exemple :

```
int tableau[] = {10,20,30,40,50};
```

Le nombre d'élément de chaque lignes peut ne pas être identique :

Exemple :

```
int[][] tabEntiers = {{1,2,3,4,5,6},
                      {1,2,3,4},
                      {1,2,3,4,5,6,7,8,9}};
```

3.8.3. Le parcours d'un tableau

Exemple :

```
for (int i = 0; i < tableau.length ; i ++) { ... }
```

La variable **length** retourne le nombre d'éléments du tableau.

Pour passer un tableau à une méthode, il suffit de déclarer les paramètres dans l'en tête de la méthode

Exemple :

```
public void printArray(String texte[]){ ...
}
```

Les tableaux sont toujours transmis par référence puisque se sont des objets.

Un accès a un élément d'un tableau qui dépasse sa capacité, lève une exception du type java.lang.arrayIndexOutOfBoundsException.

3.9. Les conversions de types

Lors de la déclaration, il est possible d'utiliser un cast :

Exemple :

```
int entier = 5;
float flottant = (float) entier;
```

La conversion peut entrainer une perte d'informations.

Il n'existe pas en java de fonction pour convertir : les conversions de type ce font par des méthodes. La bibliothèque de classes API fournit une série de classes qui contiennent des méthodes de manipulation et de conversion de types élémentaires.

Classe	Role
String	pour les chaines de caractères Unicode
Integer	pour les valeurs entières (integer)
Long	pour les entiers long signés (long)
Float	pour les nombres à virgules flottante (float)
Double	pour les nombres à virgule flottante en double précision (double)

Les classes portent le même nom que le type élémentaires sur lequel elles reposent avec la première lettre en majuscule.

Ces classes contiennent généralement plusieurs constructeurs. Pour y accéder, il faut les instancier puisque de sont des objets.

Exemple :

```
String montexte;
 montexte = new String(«test»);
```

L'objet montexte permet d'accéder aux méthodes de la classe java.lang.String

3.9.1. La conversion d'un entier int en chaine de caractère String

Exemple :

```
int i = 10;
String montexte = new String();
montexte =montexte.valueOf(i);
```

valueOf est également définie pour des arguments de type boolean, long, float, double et char

3.9.2. La conversion d'une chaine de caractères String en entier int

Exemple :

```
String montexte = new String(« 10 »);
Integer nomnombre=new Integer(montexte);
int i = monnombre.intValue(); //convertion d'Integer en int
```

3.9.3. La conversion d'un entier int en entier long

Exemple :

```
int i=10;
Integer monnombre=new Integer(i);
long j=monnombre.longValue();
```

3.10. La manipulation des chaines de caractères

La définition d'un caractère :

Exemple :

```
char touche = '%';
```

La définition d'une chaine :

Exemple :

```
String texte = « bonjour »;
```

Les variables de type String sont des objets. Partout ou des constantes chaines figurent entre guillemets, le compilateur Java génère un objet de type String avec le contenu spécifié. Il est donc possible d'ecrire :

```
String texte = « Java Java Java ».replace('a','o');
```

Les chaines ne sont pas des tableaux : il faut utiliser les méthodes de la classes String d'un objet instancié pour effectuer des manipulations.

Il est impossible de modifier le contenu d'un objet String contruit à partir d'une constante. Cependant, il est possible d'utiliser les méthodes qui renvoient une chaine pour modifier le contenu de la chaine

Exemple :

```
String texte = « Java Java Java »;
texte = texte.replace('a','o');
```

Java ne fonctionne pas avec le jeu de caractères ASCII ou ANSI, mais avec Unicode (Universal Code). Ceci concerne les types char et les chaines de caractères. Le jeu de caractères Unicode code un caractère sur 2 octets. Les caractères 0 à 255 correspondent exactement au jeu de caractères ASCII

3.10.1. Les caractères spéciaux dans les chaines

Caractères spéciaux	Affichage
\'	Apostrophe
\ »	Guillemet
\\	anti slash
\t	Tabulation
\b	retour arrière (backspace)
\r	retour chariot
\f	saut de page (form feed)
\n	saut de ligne (newline)
\0ddd	caractère ASCII ddd (octal)
\xdd	caractère ASCII dd (hexadécimal)
\udddd	caractère Unicode dddd (hexadécimal)

3.10.2 L'addition de chaines

Java admet l'opérateur + comme opérateur de concaténation de chaines de caractères.

L'opérateur + permet de concatener plusieurs chaines. Il est possible d'utiliser l'opérateur +=

Exemple :

```
String texte = « »;
texte += « Hello »;
texte += « World3 »;
```

Cet opérateur sert aussi à concatener des chaines avec tous les types de bases. La variable ou constante est alors convertie en chaine et ajoutée à la précédente. La condition préalable est d'avoir au moins une chaine dans l'expression sinon le sinon ‘+' est évalué comme opérateur mathématique.

Exemple :

```
System.out.println(« La valeur de Pi est : »+Math.PI);
int duree = 121;
System.out.println(« durée = » +duree);
```

3.10.3. La comparaison de deux chaines

Il faut utiliser la méthode equals()

Exemple :

```
String texte1 = « texte 1 »;
String texte2 = « texte 2 »;
if ( texte1.equals(texte2) )...
```

3.10.4. La détermination de la longueur d'une chaine

La méthode length() permet de déterminer la longueur d'une chaine.

Exemple :

```
String texte = « texte »;
int longueur = texte.length();
```

3.10.5. La modification de la casse d'une chaine

Les méthodes Java toUpperCase() et toLowerCase() permettent respectivement d'obtenir une chaine tout en majuscule ou tout en minuscule.

Exemple :

```
String texte = « texte »;
String textemaj = texte.toUpperCase();
```

4. La programmation orientée objet

L'idée de base de la programmation orientée objet est de rassembler dans une même entité appelée objet les données et les traitements qui s'y appliquent.

Ce chapitre ce compose de plusieurs sections :

- Le concept de classe
  Cette section présente le concept et la syntaxe de la déclaration d'une classe
- Les objets
  Cette section présente la création d'un objet, sa durée de vie, le clonage d'objets, les références et la comparaison d'objets, l'objet null, les variables de classes, la variable this et l'opérateur instanceof.
- Les modificateurs d'accès
  Cette section présente les modificateurs d'accès des entités classes, méthodes et attributs ainsi que les mots clés qui permettent de qualifier ces entités
- Les propriétés ou attributs
  Cette section présente les données d'une classe : les propriétés ou attributs
- Les méthodes
  Cette section présente la déclaration d'une méthode, la transmissions de paramètres, l'emmission de messages, la surcharge, la signature d'une méthode et le polymorphisme et des méthodes particulières : les constructeurs, le destructeur et les accesseurs
- L'héritage
  Cette section présente l'héritage : son principe, sa mise en oeuvre, ces conséquences. Il présente aussi la redéfinition d'une méthode héritée et les interfaces
- Les packages
  Cette section présente la définition et l'utilisation des packages
- Les classes internes
  Cette section présente une extension du langage java qui permet de definir une classe dans une autre.
- La gestion dynamique des objets
  Cette section présente rapidement la gestion dynamique des objets grace à l'introspection

4.1. Le concept de classe

Une classe est le support de l 'encapsulation : c'est un ensemble de données et de fonctions regroupées dans une même entité. Une classe est une description abstraite d'un objet. Les fonctions qui opèrent sur les données sont appelées des méthodes. Instancier une classe consiste à créer un objet sur son modèle. Entre classe et objet il y a, en quelque sorte, le même rapport qu'entre type et variable.

java est un langage orienté objet : tout appartient à une classe sauf les variables de type primitives.

Pour accéder à une classe il faut en déclarer une instance de classe ou objet.

Une classe comporte sa déclaration, des variables et la définition de ses méthodes.

Une classe se compose en deux parties : un en−tête et un corps. Le corps peut être divisé en 2 sections : la déclaration des données et des constantes et la définition des méthodes. Les méthodes et les données sont pourvues d'attributs de visibilité qui gère leur accessibilité par les composants hors de la classe.

4.1.1. La syntaxe de déclaration d'une classe

modificateurs nom_de_classe [extends classe_mere] [implements interface] { ... }

```
ClassModifiers class ClassName [extends SuperClass] [implemenents Interfaces]
{
   // insérer ici les champs et les méthodes
}
```

Les modificateurs de classe (ClassModifiers) sont :

Modificateur	Role
abstract	la classe contient une ou des méthodes abstraites, qui n'ont pas de définition explicite. Une classe déclarée abstract ne peut pas être instanciée : il faut définir une classe qui hérite de cette classe et qui implémente les méthodes nécessaires pour ne plus être abstraite.
final	la classe ne peut pas être modifiée, sa redéfinition grace à l'héritage est interdite. Les classes déclarées final ne peuvent donc pas avoir de classes filles.
private	la classe n'est accessible qu'à partir du fichier où elle est définie
public	La classe est accessible partout

Les modificateurs abstract et final ainsi que public et private sont mutuellement exclusifs.

Le mot clé **extends** permet de spécifier une superclasse éventuelle : ce mot clé permet de préciser la classe mère dans une relation d'héritage.

Le mot clé **implements** permet de spécifier une ou des interfaces que la classe implémente. Cela permet de récupérer quelques avantages de l'héritage multiple.

L'ordre des méthodes dans une classe n'a pas d'importance. Si dans une classe, on rencontre d'abord la méthode A puis la méthode B, B peut être appelée sans problème dans A.

4.2. Les objets

Les objets contiennent des attributs et des méthodes. Les attributs sont des variables ou des objets nécessaires au fonctionnement de l'objet. En java, une application est un objet. La classe est la description d'un objet. Un objet est une instance d'une classe. Pour chaque instance d'une classe, le code est le même, seul les données sont différentes à chaque objet.

4.2.1. La création d'un objet : instancier une classe

Il est nécessaire de définir la déclaration d'une variable ayant le type de l'objet désiré. La déclaration est de la forme classe nom_de_variable

Exemple :
`MaClasse m;`

L'opérateur new se charge de créer une instance de la classe et de l'associer à la variable

Exemple :

```
m = new MaClasse();
```

Il est possible de tout reunir en une seule déclaration

Exemple :

```
MaClasse m = new MaClasse();
```

Chaque instance d'une classe nécessite sa propre variable. Plusieurs variables peuvent désigner un même objet.

En Java, tous les objets sont instanciés par allocation dynamique. Dans l'exemple, la variable m contient une référence sur l'objet instancié (contient l'adresse de l'objet qu'elle désigne : attention toutefois, il n'est pas possible de manipuler ou d ‘effectuer des opérations directement sur cette adresse comme en C).

Si m2 désigne un objet de type MaClasse, l'instruction m2 = m ne définit pas un nouvel objet mais m et m2 désignent tous les deux le même objet.

L'opérateur new est un opérateur de haute priorité qui permet d'instancier des objets et d'appeler une méthode particulière de cet objet : le contructeur. Il fait appel à la machine virtuelle pour obtenir l'espace mémoire nécessaire à la représentation de l'objet puis appelle le constructeur pour initialiser l'objet dans l'emplacement obtenu. Il renvoie une valeur qui référence l'objet instancié.

Si l'opérateur new n'obtient pas l'allocation mémoire nécessaire il lève l'exception OutOfMemoryError.

Remarque sur les objets de type String : Un objet String est automatiquement créer lors de l'utilisation d'une constante chaine de caractères sauf si celle ci est déjà utilisée dans la classe. Ceci permet une simplification dans l'écriture des programmes.

Exemple :

```
String chaine = « bonjour »
et String chaine = new String(« bonjour »)
```

sont équivalents.

4.2.2. La durée de vie d'un objet

Les objets ne sont pas des éléments statiques et leur durée de vie ne correspond pas forcément à la durée d'exécution du programme.

La durée de vie d'un objet passe par trois étapes :

- la déclaration de l'objet et l'instanciation grace à l'opérateur new

  Exemple :

  ```
  nom_de_classe nom_d_objet = new nom_de_classe( ... );
  ```

- l'utilisation de l'objet en appelant ces méthodes
- la suppression de l'objet : elle est automatique en java grace à la machine virtuelle. La restitution de la mémoire inutilisée est prise en charge par le récupérateur de mémoire (garbage collector). Il n'existe pas d'instruction delete comme en C++.

4.2.3. La création d'objets identiques

Exemple :

```
MaClasse m1 = new MaClasse();
MaClasse m2 = m1;
```

m1 et m2 contiennent la même référence et pointent donc tous les deux sur le même objet : les modifications faites à partir d'une des variables modifient l'objet.

Pour créer une copie d'un objet, il faut utiliser la méthode clone() : cette méthode permet de créer un deuxième objet indépendant mais identique à l'original. Cette méthode est héritée de la classe Object qui est la classe mère de toute les classes en Java.

Exemple :

```
MaClasse m1 = new MaClasse();
MaClasse m2 = m1.clone();
```

m1 et m2 ne contiennent plus la même référence et pointent donc sur des objets différents.

4.2.4. Les références et la comparaison d'objets

Les variables de type objet que l'on déclare ne contiennent pas un objet mais une référence vers cette objet. Lorque l'on écrit c1 = c2 (c1 et c2 sont des objets), on copie la référence de l'objet c2 dans c1 : c1 et c2 réfèrent au même objet (ils pointent sur le même objet). L'opérateur == compare ces références. Deux objets avec des propriétés identiques sont deux objets distincts :

Exemple :

```
Rectangle r1 = new Rectangle(100,50);
Rectangle r2 = new Rectangle(100,50);
if (r1 == r1) { ... } // vrai
if (r1 == r2) { ... } // faux
```

Pour comparer l'égalité des variables de deux instances, il faut munir la classe d'un méthode à cette effet : la méthode equals héritée de Object.

Pour s'assurer que deux objets sont de la même classe, il faut utiliser la méthode getClass() de la classe Object dont toutes les classes héritent.

Exemple :

```
(obj1.getClass().equals(obj2.getClass())
```

4.2.5. L'objet null

L'objet null est utilisable partout. Il n'appartient pas à une classe mais il peut être utilisé à la place d'un objet de n'importe quelle classe ou comme paramètre. null ne peut pas etre utilisé comme un objet normal : il n'y a pas d'appel de méthodes et aucunes classes ne peut en hériter.

Le fait d'initialiser un variable référent un objet à null permet au ramasse miette de libérer la mémoire allouée à l'objet.

4.2.6. Les variables de classes

Elles ne sont définies qu'une seule fois quelque soit le nombre d'objets instanciés de la classe. Leur déclaration est accompagnée du mot clé static

Exemple :

```
public class MaClasse() {
    static int compteur = 0;
}
```

L'appartenance des variables de classe à une classe entière et non à un objet spécifique permet de remplacer le nom de la variable par le nom de la classe.

Exemple :

```
MaClasse m = new MaClasse();
int c1 = m.compteur;
int c2 = MaClasse.compteur;
```

c1 et c2 possèdent la même valeur.

Ce type de variable est utile pour par exemple compter le nombre d'instanciation de la classe qui est faite.

4.2.7. La variable this

Cette variable sert à référencer dans une méthode l'instance de l'objet en cours d'utilisation. this est un objet qui est égale à l'instance de l'objet dans lequel il est utilisé.

Exemple :

```
private int nombre;
public maclasse(int nombre) {
   nombre = nombre; // variable de classe = variable en paramètre du constructeur
}
```

Il est préférable d'écrire

```
this.nombre = nombre;
```

Cette référence est habituellement implicite :

Exemple :

```
class MaClasse() {
  String chaine = « test » ;
  Public String getChaine() { return chaine) ;
  // est équivalent à public String getChaine (this.chaine);
}
```

This est aussi utilisé quand l'objet doit appeler une méthode en se passant lui même en paramètre de l'appel.

4.2.8. L'opérateur instanceof

L'opérateur instanceof permet de déterminer la classe de l'objet qui lui est passé en paramètre. La syntaxe est objet instanceof classe

Exemple :

```
void testClasse(Object o) {
   if (o instanceof MaClasse )
      System.out.println(« o est une instance de la classe MaClasse »);
   else System.out.println(« o n'est pas un objet de la classe MaClasse »);
}
```

Il n'est toutefois pas possible d'appeler une méthode de l'objet car il est passé en paramètre avec un type Object

Exemple :

```
void afficheChaine(Object o) {
    if (o instanceof MaClasse)
        System.out.println(o.getChaine());
        // erreur à la compil car la méthode getChaine()
        //n'est pas définie dans la classe Object
}
```

Pour résoudre le problème, il faut utiliser la technique du casting (conversion).

Exemple :

```
void afficheChaine(Object o) {
   if (o instanceof MaClasse)
   {
      MaClasse m = (MaClasse) o;
      System.out.println(m.getChaine());
      // OU System.out.println( ((MaClasse) o).getChaine() );
   }
}
```

4.3. Les modificateurs d'accès

Ils se placent avant ou après le type de l'objet mais la convention veut qu'ils soient placés avant.

Ils s'appliquent aux classes et/ou aux méthodes et/ou aux attributs.

Ils ne peuvent pas être utilisés pour qualifier des variables locales : seules les variables d'instances et de classes peuvent en profiter.

Ils assurent le contrôle des conditions d'héritage, d'accès aux éléments et de modification de données par les autres objets.

4.3.1. Les mots clés qui gèrent la visibilité des entités

De nombreux langages orientés objet introduisent des attributs de visibilité pour reglémenter l'accès aux classes et aux objets, aux méthodes et aux données.

Il existe 3 modificateurs qui peuvent être utilisés pour définir les attributs de visibilité des entités (classes, méthodes ou attributs) : public, private et protected. Leur utilisation permet de définir des niveaux de protection différents (présenté dans un ordre croissant de niveau de protection offert):

Modificateur	Role
public	Une variable, méthode ou classe déclarée public est visible par tout les autres objets. Dans la version 1.0, une seule classe public est permise par fichier et son nom doit correspondre à celui du fichier. Dans la philosophie orientée objet aucune donnée d'une classe ne devraient être déclarée publique : il est préférable d'écrire des méthodes pour la consulter et la modifier
par défaut : package friendly	Il n'existe pas de mot clé pour définir ce niveau, qui est le niveau par défaut lorsqu'aucun modificateur n'est précisé. Cette déclaration permet à une entité (classe, méthode ou variable) d'être visible par toutes les classes se trouvant dans le même package.
protected	Si une classe, une méthode ou une variable est déclarée protected , seules les méthodes présentes dans le même package que cette classe ou ses sous classes pourront y acceder. On ne peut pas qualifier une classe avec protected.
private	C'est le niveau de protection le plus fort. Les composants ne sont visibles qu'à l'intérieur de la classe : ils ne peuvent être modifiés que par des méthodes définies dans la classe prévues à cet effet. Les méthodes déclarée private ne peuvent pas être en même temps déclarée abstract car elles ne peuvent pas être redéfinies dans les classes filles.

Ces modificateurs d'accès sont mutuellement exclusifs.

4.3.2. Le mot clé static

Le mot clé static s'applique aux variables et aux méthodes.

Les variables d'instance sont des variables propres à un objet. Il est possible de définir une variable de classe qui est partagée entre toutes les instances d'une même classe : elle n'existe donc qu'une seule fois. Une telle variable permet de stocker une constante ou une valeur modifiée tour à tour par les instances de la classe. Elle se définit avec le mot clé static.

Exemple :

```
public class Cercle {
    static float pi = 3.1416f;
    float rayon;
    public Cercle(float rayon) { this.rayon = rayon; }
    public float surface() { return rayon * rayon * pi;}
}
```

Il est aussi possible par exemple de mémoriser les valeurs min et max d'un ensemble d'objet de même classe.

Une méthode static est une méthode qui n'agit pas sur des variables d'instance mais uniquement sur des variables de classe. Les méthodes ainsi définies peuvent être appelée avec la notation classe.méthode au lieu de objet.méthode. Ces méthodes peuvent être utilisées sans instancier un objet de la classe.

Il n'est pas possible d'appeler une méthode d'instance ou d'accéder à une variable d'instance à partir d'une méthode de classe statique

4.3.3. Le mot clé final

Le mot cle final s'applique aux variables, aux méthodes et aux classes.

Une variable qualifiée de final signifie que la variable est constante. Le compilateur ne contrôle ni n'empéche la modification. On ne peut déclarer de variables final locale à une méthode. Les constantes sont qualifiées de final et static

Exemple :

```
public static final float PI = 3.1416f;
```

Une méthode final ne peut pas être redéfinie dans une sous classe. Une méthode possédant le modificateur final pourra être optimisée par le compilateur car il est garanti qu'elle ne sera pas sous classée.

Lorsque le modificateur final est ajouté à une classe, il est interdit de créer un classe qui en hérite.

Pour une méthode ou une classe, on renonce à l'héritage mais ceci peut s'avérer nécessaire pour des questions de sécurité ou de performance. Le test de validité de l'appel d'une méthode est bien souvent repoussé à l'éxécution, en fonction du type de l'objet appelé (c'est le polymorphisme). Ces tests ont un coup en terme de performance.

4.3.4. Le mot clé abstract

Le mot cle abstract s'applique aux méthodes et aux classes.

Abstract indique que la classe ne pourra être instanciée telle quelle. De plus, toutes les méthodes de cette classe abstract ne sont pas implémentées et devront être redéfinies par des méthodes complètes dans ses sous classes.

Abstract permet de créer une classe qui sera une sorte de moule. Toutes les classes dérivées pouront profiter des méthodes héritées et n'auront à implémenter que les méthodes déclarées abstract.

Exemple :

```
abstract class ClasseAbstraite {
    ClasseBastraite() { ... //code du constructeur }
    void méthode() { ... // code partagé par tous les descendants}
    abstract void méthodeAbstraite();
}

class ClasseComplete extends ClasseAbstraite {
    ClasseComplete() { super(); … }
    void méthodeAbstraite() { ... // code de la méthode }
    // void méthode est héritée
}
```

Une méthode abstraite est une méthode déclarée avec le modificateur abstract et sans corps. Elle correspond à une méthode dont on veut forcer l'implémentation dans une sous classe. L'abstraction permet une validation du codage : une sous classe sans le modificateur abstract et sans définition explicite d'une ou des méthodes abstraites génère une erreur de compilation.

Une classe est automatiquement abstraite dès qu'une de ses méthodes est déclarée abstraite. Il est possible de définir une classe abstraite sans méthodes abstraites.

4.3.5. Le mot clé synchronized

Permet de gérer l'accès concurrent aux variables et méthodes lors de traitement de thread (exécution « simultanée » de plusieurs petites parties de code du programme)

4.3.6. Le mot clé volatile

Le mot cle volatile s'applique aux variables.

Précise que la variable peut être changée par un périphérique ou de manière asynchrone. Cela indique au compilateur de ne pas stocker cette variable dans des registres. A chaque utilisation, on lit la valeur et on réécrit immédiatement le résultat s'il a changé.

4.3.7. Le mot clé native

Une méthode native est une méthode qui est implémentée dans un autre langage. L'utilisation de ce type de méthode limite la portabilité du code mais permet une vitesse exécution plus rapide.

4.4. Les propriétés ou attributs

Les données d'une classe sont contenues dans les propriétés ou attributs. Ce sont des variables qui peuvent être des variables d'instances, des variables de classes ou des constantes.

4.4.1. Les variables d'instances

Une variable d'instance nécessite simplement une déclaration de la variable dans le corps de la classe.

Exemple :

```
public class MaClasse {
   public int valeur1 ;
   int valeur2 ;
   protected int valeur3 ;
   private int valeur4 ;
}
```

Chaque instance de la classe a accès à sa propre occurrence de la variable.

4.4.2. Les variables de classes

Les variables de classes sont définies avec le mot clé static

Exemple (code java 1.1) :

```
public class MaClasse {
   static int compteur ;
}
```

Chaque instance de la classe partage la même variable.

4.4.3. Les constantes

Les constantes sont définies avec le mot clé final : leur valeur ne peut pas être modifiée.

Exemple (code java 1.1) :

```
public class MaClasse {
   final double pi=3.14 ;
}
```

4.5. Les méthodes

Les méthodes sont des fonctions qui implémentent les traitements de la classe.

4.5.1. La syntaxe de la déclaration

La syntaxe de la déclaration d'une méthode est :

```
modificateurs type_retourné nom_méthode ( arg1, ... ) {... }
    // définition des variables locales et du bloc d'instructions
}
```

Le type retourné peut être élémentaire ou correspondre à un objet. Si la méthode ne retourne rien, alors on utilise **void**

Le type est le nombre d'arguments déclarés doivent correspondrent au type et au nombre d'arguments transmis. Il n'est pas possible d'indiquer des valeurs par défaut dans les paramètres. Les arguments sont passés par valeur : la méthode fait une copie de la variable qui lui est locale. Lorsqu'un objet est transmis comme argument à une méthode, cette dernière reçoit une référence qui désigne son emplacement mémoire d'origine et qui est une copie de la variable . Il est possible de modifier l'objet grace à ces méthodes mais il n'est pas possible de remplacer la référence contenue dans la variable passée en paramètre : ce changement n'aura lieu que localement à la méthode.

Les modificateurs de méthodes sont :

Modificateur	Role
public	la méthode est accéssible aux méthodes des autres classes
private	l'usage de la méthode est réservée aux autres méthodes de la même classe
protected	la méthode ne peut être invoquée que par des méthodes de la classe ou de ses sous classes
final	la méthode ne peut être modifiée (redifinition lors de l'héritage interdite)
static	la méthode appartient simultanément à tous les objets de la classe (comme une constante déclarée à l'intérieur de la classe). Il est inutile d'instancier la classe pour appeler la méthode mais la méthode ne peut pas manipuler de variable d'instance. Elle ne peut utiliser que des variables de classes.
synchronized	la méthode fait partie d'un thread. Lorsqu'elle est appelée, elle barre l'accès à son instance. L'instance est à nouveau libérée à la fin de son éxécution.
native	le code source de la méthode est écrit dans un autre langage

Sans modificateur, la méthode peut être appelée par toutes autres méthodes des classes du package auquel appartient la

classe.

La valeur de retour de la méthode doit être transmise par l'instruction return. Elle indique la valeur que prend la méthode et termine celle ci : toutes les instructions qui suivent return sont donc ignorées.

Exemple :

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

Il est possible d'inclure une instruction return dans une méthode de type void : cela permet de quitter la méthode.

La méthode main() de la classe principale d'une application doit être déclarée de la façon suivante :

Déclaration d'une méthode main() :

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

Si la méthode retourne un tableau alors les [] peuvent être préciser après le type de retour ou après la liste des paramètres :

Exemple :

```
int[] valeurs() { … }
int valeurs()[] { … }
```

4.5.2. La transmission de paramètres

Lorsqu'un objet est passé en paramètre, ce n'est pas l'objet lui même qui est passé mais une référence sur l'objet. La référence est bien transmise par valeur et ne peut pas être modifiée mais l'objet peut être modifié via un message (appel d'une méthode).

Pour transmettre des arguments par référence à une méthode, il faut les encapsuler dans un objet qui prévoit les méthodes nécessaires pour les mises à jour.

Si un objet o transmet sa variable d'instance v en paramètre à une méthode m, deux situations sont possibles :

- si v est une variable primitive alors elle est passée par valeur : il est impossible de la modifier dans m pour que v en retour contiennent cette nouvelle valeur.
- si v est un objet alors m pourra modifier l'objet en utilisant une méthode de l'objet passé en paramètre.

4.5.3. L'emmission de messages

Un message est émis lorsqu'on demande à un objet d'exécuter l'une de ses méthodes.

La syntaxe d'appelle d'une méthode est : nom_objet.nom_méthode(parametre, …) ;

Si la méthode appelée ne contient aucun paramètre, il faut laisser les parenthèses vides.

4.5.4. L'enchainement de références à des variables et à des méthodes

Exemple :

```
System.out.Println("bonjour");
```

Deux classes sont impliqués dans l'instruction : System et PrintStream. La classe System possède une variable nommée out qui est un objet de type PrintStream. Println() est une méthode de la classe PrintStream. L'instruction signifie : « utilise la méthode Println() de la variable out de la classe System ».

4.5.5. La surcharge de méthodes

La surcharge d'une méthode permet de définir plusieurs fois une même méthode avec des arguments différents. Le compilateur choisi la méthode qui doit être appellée en fonction du nombre et du type des arguments. Ceci permet de simplifier l'interface des classes vis à vis des autres classes.

Une méthode est surchargée lorqu'elle exécute des actions différentes selon le type et le nombre de paramètres transmis.

Exemple :

```
class affiche{
    public void afficheValeur(int i) {
        System.out.pintln(« nombre entier = » + i);
    }

    public void afficheValeur(float f) {
        System.out.println(« nombre flottant = » + f);
    }
}
```

Il n'est pas possible d'avoir deux méthodes de même nom dont tous les paramètres sont identiques et dont seul le type retourné diffère.

Exemple :

```
class Affiche{

    public float convert(int i){
         return((float) i);
    }

    public double convert(int i){
        return((double) i);
    }
}
```

Résultat à la compilation :

```
C:\>javac Affiche.java
Affiche.java:5: Methods can't be redefined with a different return type: double
convert(int) was float convert(int)
public double convert(int i){
       ^
1 error
```

4.5.6. La signature des méthodes et le polymorphisme

Il est possible de donner le même nom à deux méthodes différentes à condition que les signatures de ces deux méthodes soient différentes. La signature d'une méthode comprend le nom de la classe, le nom de la méthode et les types des paramètres. Cette facilité permet de mettre en oeuvre la polymorphisme.

Le polymorphisme est la capacité, pour un même message de correspondre à plusieurs formes de traitements selon l'objet auquel ce message est adressé. La gestion du polymorphisme est assurée par la machine virtuelle dynamiquement à l'exécution

4.5.7. Les constructeurs

La déclaration d'un objet est suivie d'une sorte d'initialisation par le moyen d'une méthode particulière appelée constructeur pour que les variables aient une valeur de départ. Elle n'est systématiquement invoquée que lors de la création d'un objet.

Le constucteur suit la définition des autres méthodes excepté que son nom doit obligatoirement correspondre à celui de la classe et qu'il n'est pas typé, pas même void, donc il ne peut pas y avoir d'instruction return dans un constructeur. On peut surcharger un constructeur.

La définition d'un constructeur est facultative. Si elle n'est pas définie, la machine virtuelle appelle un constructeur par défaut vide créé automatiquement. Dès qu'un constructeur est explicitement défini, Java considère que le programmeur prend en charge la création des constructeurs et que le mécanisme par défaut, qui correspond à un constructeur sans paramètres, est supprimé. Si on souhaite maintenir ce mécanisme, il faut définir explicitement un constructeur sans paramètres.

Il existe plusieurs manière de définir un constructeur :

1. le constructeur simple : ce type de constructeur ne nécessite pas de définition explicite : son existence découle automatiquement de la définition de la classe.

Exemple :

```
public MaClasse() {}
```

2. le constructeur avec initialisation fixe : il permet de créer un constructeur par défaut

Exemple :

```
public MaClasse() {
    nombre = 5;
}
```

3. le constructeur avec initialisation des variables : pour spécifier les valeurs de données à initialiser on peut les passer en paramètres au constructeur

Exemple :

```
public MaClasse(int valeur) {
     nombre = valeur;
}
```

4.5.8. Le destructeur

Un destructeur permet d'éxécuter du code lors de la libération de la place mémoire occupée par l'objet. En java, les destructeurs appellés finaliseurs (finalizers), sont automatiquement appellés par le garbage collector.

Pour créer un finaliseur, il faut redéfinir la méthode finalize() héritée de la classe Object.

4.5.9. Les accesseurs

L'encapsulation permet de sécuriser l'accès aux données d'une classe. Ainsi, les données déclarées private à l'intérieur d'une classe ne peuvent être accédées et modifiées que par des méthodes définies dans la même classe. Si une autre classe veut accéder aux données de la classe, l'opération n'est possible que par l'intermédiaire d'une méthode de la classe prévue à cet effet. Ces appels de méthodes sont appelés « échanges de message ».

Un accesseur est une méthode publique qui donne l'accès à une variable d'instance privée. Pour une variable d'instance, il peut ne pas y avoir d'accesseur, un seul accesseur en lecture ou un accesseur en lecture et un en écriture. Par convention, les accesseurs en lecture commencent par get et les accesseurs en écriture commencent par set.

Exemple :

```
private int valeur = 13;

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

public void setValeur(int val) {
    valeur = val;
}
```

4.6. L'héritage

4.6.1. Le principe de l'héritage

L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution.

Grace à l'heritage, les objets d'une classe ont accès aux données et aux méthodes de la classe parent et peuvent les étendre. Les sous classes peuvent redéfinir les variables et les méthodes héritées. Pour les variables, il suffit de les redéclarer sous le même nom avec un type différent. Les méthodes sont redéfinies avec le même nom, les mêmes types et le même nombre d'arguments, sinon il s'agit d'une surcharge.

L'héritage successif de classes permet de définir une hiérarchie de classe qui ce compose de super classes et de sous classes. Une classe qui hérite d'une autre est une sous classe et celle dont elle hérite est une super classe. Une classe peut avoir plusieurs sous classes. Une classe ne peut avoir qu'une seule classe mère : il n'y a pas d'héritage multiple en java.

Object est la classe parente de toutes les classes en java. Toutes les variables et méthodes contenues dans Object sont accessibles à partir de n'importe quelle classe car par héritage succéssif toutes les classes héritent d'Object.

4.6.2. La mise en oeuvre de l'héritage

On utilise le mot clé extends pour indiquer qu'une classe hérite d'une autre. En l'absence de ce mot réservé associé à une classe, le compilateur considère la classe Object comme classe parent.

Exemple :

```
class Fille extends Mere { ... }
```

Pour invoquer une méthode d'une classe parent, il suffit d'indiquer la méthode préfixée par super. Pour appeler le constructeur de la classe parent il suffit d'écrire super(paramètres) avec les paramètres adéquats.

Le lien entre une classe fille et une classe parent est géré par le langage : une évolution des règles de gestion de la classe parent conduit à modifier automatiquement la classe fille dès que cette dernière est recompilée.

En java, il est obligatoire dans un constructeur d'une classe fille de faire appel explicitement ou implicitement au constructeur de la classe mère.

4.6.3. L'accès aux propriétés héritées

Les variables et méthodes définies avec le modificateur d'accès public restent publiques à travers l'héritage et toutes les autres classes.

Une variable d'instance définie avec le modificateur private est bien héritée mais elle n'est pas accessible directement mais via les méthodes héritées.

Si l'on veut conserver pour une variable d'instance une protection semblable à celle assurée par le modificateur private, il faut utiliser le modificateur protected. La variable ainsi définie sera hérité dans toutes les classes descendantes qui pourront y accéder librement mais ne sera pas accessible hors de ces classes directement.

4.6.4 Le transtypage induit par l'héritage facilitent le polymorphisme

L'héritage définit un cast implicite de la classe fille vers la classe mere : on peut affecter à une référence d'une classe n'importe quel objet d'une de ses sous classes.

Exemple : la classe Employe hérite de la classe Personne

```
    Personne p = new Personne («Dupond», «Jean»);
    Employe e = new Employe(«Durand», «Julien», 10000);
    p = e ; // ok : Employe est une sous classe de Personne
    Objet obj;
    obj = e ; // ok : Employe herite de Personne qui elle même hérite de Object
```

Il est possible d'écrire le code suivant si Employe hérite de Personne

Exemple :

```
    Personne[] tab = new Personne[10];
    tab[0]:= new Personne( «Dupond»,«Jean»);
    tab[1]:= new Employe(«Durand», «Julien», 10000);
```

Il est possible de surcharger une méthode héritée : la forme de la méthode à éxécuter est choisie en fonction des paramètres associés à l'appel.

Compte tenu du principe de l'héritage, le temps d'exécution du programme et la taille du code source et de l'éxécutable augmentent.

4.6.5. La redéfinition d'une méthode héritée

La redéfinition d'une méthode héritée doit impérativement conserver la déclaration de la méthode parent (type et nombre de paramètres, la valeur de retour et les exceptions propagées doivent être identique).

Si la signature de la méthode change, ce n'est plus une redéfinition mais une surcharge. Cette nouvelle méthode n'est pas héritée : la classe mere ne possède pas de méthode possédant cette signature.

4.6.6. Les interfaces et l'héritage multiple

Avec l'héritage multiple, une classe peut hériter en même temps de plusieurs super classes. Ce mécanisme n'existe pas en java. Les interfaces permettent de mettre en œuvre un mécanisme de remplacement.

Une interface est un ensemble de constantes et de déclarations de méthodes correspondant un peu à une classe abstraite. C'est une sorte de standard auquel une classe peut répondre. Tous les objets qui se conforment à cette interface (qui implémentent cette interface) possèdent les méthodes et les constantes déclarée dans celle−ci. Plusieurs interfaces peuvent être implémentées dans une même classe.

Les interfaces se déclarent avec le mot cle interface et sont intégrées aux autres classes avec le mot clé implements. Une interface est implicitement déclarée avec le modificateur abstract.

Déclaration d'une interface :

```
[public] interface nomInterface [extends nomInterface1, nomInterface2 ... ] {
    // insérer ici des méthodes ou des champs static
}
```

Implémentation d'une interface :

```
Modificateurs class nomClasse [extends superClasse]
   [implements nomInterface1, nomInterface 2, ...] {
    //insérer ici des méthodes et des champs

}
```

Exemple :

```
interface AfficheType {
    void AfficheType();
}

class Personne implements AfficheType {

    public void afficheType() {
        System.out.println(« Je suis une personne »);
    }
}

class Voiture implements AfficheType {

    public void afficheType() {

        System.out.println(« Je suis une voiture »);
    }
}
```

Exemple : déclaration d'un interface à laquelle doit se conformer tout individus

```
interface Individu {
    String getNom();
    String getPrenom();
    Date getDateNaiss();
}
```

Toutes les méthodes d'une interface sont publiques et abstraites : elles sont implicitement déclarées comme telles.

Une interface peut être d'accès public ou package. Si elle est publique, toutes ses méthodes sont publiques même si elles ne sont pas déclarées avec le modificateur public. Si elle est d'accès package, il s'agit d'une interface d'implémentation pour les autres classes du package et ses méthodes ont le même accès package : elles sont accessible à toutes les classes du packages.

Les seules variables que l'on peut définir dans une interface sont des variables de classe qui doivent être constantes : elles sont donc implicitement déclarées avec le modificateur static et final même si elles sont définies avec d'autres modificateurs.

Toute classe qui implémente cette interface doit au moins posséder les méthodes qui sont déclarées dans l'interface. L'interface ne fait que donner une liste de méthodes qui seront à définir dans les classes qui implémentent l'interface.

Les méthodes déclarées dans une interface publique sont implicitement publiques et elles sont héritées par toutes les classes qui implémentent cette interface. Une tel classe doit, pour être instanciable, définir toutes les méthodes héritées de l'interface.

Une classe peut implémenter une ou plusieurs interfaces tout en héritant de sa classe mère.

L'implémentation d'une interface définit un cast : l'implémentation d'une interface est une forme d'héritage. Comme pour l'héritage d'une classe, l'héritage d'une classe qui implémente une interface définit un cast implicite de la classe fille vers cette interface. Il est important de noter que dans ce cas il n'est possible de faire des appels qu'à des méthodes de l'interface. Pour utiliser des méthodes de l'objet, il faut définir un cast explicite : il est préférable de controler la classe de l'objet pour éviter une exception ClassCastException à l'éxecution

4.6.7. Des conseils sur l'héritage

Lors de la création d'une classe « mère » il faut tenir compte des points suivants :

- la définition des accès aux variables d'instances, très souvent privées, doit être réfléchie entre protected et private
- pour empecher la redéfinition d'une méthode (surcharge) il faut la déclarer avec le modificateur final

Lors de la création d'une classe fille, pour chaque méthode héritée qui n'est pas final, il faut envisager les cas suivant :

- la méthode héritée convient à la classe fille : on ne doit pas la redéfinir
- la méthode héritée convient mais partiellement du fait de la spécialisation apportée par la classe fille : il faut la redéfinir voir la surcharger. La plupart du temps une redéfinition commencera par appeler la méthode héritée (via super) pour garantir l 'évolution du code
- la méthode héritée ne convient pas : il faut redéfinir ou surcharger la méthode sans appeler la méthode héritée lors de la redéfinition.

4.7. Les packages

4.7.1. La définition d'un package

En java, il existe un moyen de regrouper des classe voisines ou qui couvrent un même domaine : ce sont les packages. Pour réaliser un package, on écrit un nombre quelconque de classes dans plusieurs fichiers d'un même repertoire et au début de chaque fichier on met la directive ci dessous ou nom−du−package doit être identique au nom du répertoire :

package nomPackage;

La hiérarchie d'un package se retrouve dans l'arborescence du disque dur puisque chaque package est dans un répertoire nommé du nom du package.

Remarque : Il est préférable de laisser les fichiers source .java avec les fichiers compilés .class

D'une façon générale, l'instruction package associe toutes les classes qui sont définies dans un fichier source à un même package.

Le mot clé package doit être la première instruction dans un fichier source et il ne doit être présent qu'une seule fois dans le fichier source (une classe ne peut pas appartenir à plusieurs packages).

4.7.2. L'utilisation d'un package

Pour utiliser ensuite le package ainsi créé, on l'importe dans le fichier :

import nomPackage.*;

Pour importer un package, il y a trois méthodes si le chemin de recherche est correctement renseigné :

Exemple	Role
`import nomPackage;`	les classes ne peuvent pas être simplement désignées par leur nom et il faut aussi préciser le nom du package
`import nomPackage.*;`	toutes les classes du package sont importées
`import nomPackage.nomClasse;`	appel à une seule classe : l'avantage de cette notation est de réduire le temps de compilation

Attention : l'astérisque n'importe pas les sous paquetages. Par exemple, il n'est pas possible d'écrire import java.*.

Il est possible d'appeler une méthode d'un package sans inclure ce dernier dans l'application en precisant son nom complet :

nomPackage.nomClasse.nomméthode(arg1, arg2 ...)

Il existe plusieurs types de packages : le package par défaut (identifié par le point qui représente le répertoire courant et permet de localiser les classes qui ne sont pas associées à un package particulier), les packages standards qui sont empaquetés dans le fichier classes.zip et les packages personnels

Le compilateur implémente automatiquement une commande import lors de la compilation d'un programme Java même si elle ne figure pas explicitement au début du programme : import java.lang.*; Ce package contient entre autre les classes de base de tous les objets java dont la classe Object.

Un package par défaut est systématiquement attribué par le compilateur aux classes qui sont définies sans déclarer explicitement une appartenance à un package. Ce package par défaut correspond au répertoire courant qui est le

répertoire de travail.

4.7.3. La collision de classes.

Deux classes entre en collision lorsqu'elles portent le même nom mais qu'elles sont définies dans des packages différents. Dans ce cas, il faut qualifier explicitement le nom de la classe avec le nom complet du package.

4.7.4. Les packages et l'environnement système

Les classes Java sont importées par le compilateur (au moment de la compilation) et par la machine virtuelle (au moment de l'éxécution). Les techniques de chargement des classes varient en fonction de l'implémentation de la machine virtuelle. Dans la plupart des cas, une variable d'environnement CLASSPATH référence tous les répertoires qui hébergent des packages succeptibles d'être importés.

Exemple sous Windows :

CLASSPATH = .;C:\Java\JDK\Lib\classes.zip; C:\rea_java\package

L'importation des packages ne fonctionne que si le chemin de recherche spécifié dans une variable particulière pointe sur les packages, sinon le nom du package devra refléter la structure du répertoire ou il se trouve. Pour déterminer l'endroit où se trouvent les fichiers .class à importer, le compilateur utilise une variable d'environnement dénommée CLASSPATH. Le compilateur peut lire les fichiers .class comme des fichiers indépendants ou comme des fichiers ZIP dans lesquels les classes sont reunies et compressées.

4.8. Les classes internes

Les classes internes (inner classes) sont une extension du langage java introduite dans la version 1.1 du JDK. Ce sont des classes qui sont définies dans une autre classe. Les difficultés dans leur utilisation concerne leur visibilité et leur accès aux membres de la classe dans laquelle elles sont définies.

Exemple très simple :

```
public class ClassePrincipale1 {
  class ClasseInterne {
  }
}
```

Les classes internes sont particulièrement utiles pour :

- permettre de définir une classe à l'endroit ou une seule autre en a besoin
- définir des classes de type adapter (essentiellement à partir du JDK 1.1 pour traiter des évenements émis par les interfaces graphiques)
- définir des méthodes de type callback d'une façon générale

Pour permettre de garder une compatibilité avec la version précédente de la JVM, seul le compilateur a été modifié. Le compilateur interprète la syntaxe des classes internes pour modifier le code source et générer du byte code compatible avec la première JVM.

Il est possible d'imbriquer plusieurs classes internes. Java ne possèdent pas de restrictions sur le nombre de classes qu'il est ainsi possible d'imbriquer. En revanche une limitation peut intervenir au niveau du système d'exploitation en ce qui concerne la longueur du nom du fichier .class généré pour les différentes classes internes.

Si plusieurs classes internes sont imbriquées, il n'est pas possible d'utiliser un nom pour la classe qui soit déjà attribuée à une de ces classes englobantes. Le compilateur génèrera une erreur à la compilation.

Exemple :

```
public class ClassePrincipale6 {
  class ClasseInterne1 {
    class ClasseInterne2 {
      class ClasseInterne3 {
      }
    }
  }
}
```

Le nom de la classe interne utilise la notation qualifié avec le point préfixé par le nom de la classe principale. Ainsi, pour utiliser ou accéder à une classe interne dans le code, il faut la préfixer par le nom de la classe principale suivi d'un point.

Cependant cette notation ne représente pas physiquement le nom du fichier qui contient le byte code. Le nom du fichier qui contient le byte code de la classe interne est modifié par le compilateur pour éviter des conflits avec d'autres nom d'entité : à partir de la classe principale, les points de séparation entre chaque classe interne sont remplacé par un caractère $ (dollard).

Par exemple, la compilation du code de l'exemple précédent génère quatres fichiers contenant le byte code :
ClassePrincipale6$ClasseInterne1$ClasseInterne2ClasseInterne3.class
ClassePrincipale6$ClasseInterne1$ClasseInterne2.class
ClassePrincipale6$ClasseInterne1.class
ClassePrincipale6.class

L'utilisation du signe $ entre la classe principale et la classe interne permet d'éviter des confusions de nom entre le nom d'une classe appartenant à un package et le nom d'une classe interne.

L'avantage de cette notation est de créer un nouvel espace de nommage qui dépend de la classe et pas d'un package. Ceci renforce le lien entre la classe interne et sa classe englobante.

C'est le nom du fichier qu'il faut préciser lorsque l'on tente de charger la classe avec la méthode forName() de la classe Class. C'est aussi sous cette forme qu'est restitué le résultat d'un appel aux méthodes getClass().getName() sur un objet qui est une classe interne.

Exemple :

```
public class ClassePrincipale8 {
  public class ClasseInterne {
  }

  public static void main(String[] args) {
    ClassePrincipale8 cp = new ClassePrincipale8();
    ClassePrincipale8.ClasseInterne ci = cp. new ClasseInterne() ;
    System.out.println(ci.getClass().getName());
  }
}
```

Resultat :

```
java ClassePrincipale8
ClassePrincipale8$ClasseInterne
```

L'accessibilité à la classe interne respecte les règles de visibilité du langage. Il est même possible de définir une classe interne private pour limiter son accès à sa seule classe principale.

Exemple :

```
public class ClassePrincipale7 {
        private class ClasseInterne {
        }
}
```

Il n'est pas possible de déclarer des membres statiques dans une classe interne :

Exemple :

```
public class ClassePrincipale10 {
        public class ClasseInterne {
                static int var = 3;
        }
}
```

Resultat :

```
javac ClassePrincipale10.java
ClassePrincipale10.java:3: Variable var can't be static in inner class ClassePri
ncipale10. ClasseInterne.  Only members of interfaces and top-level classes can
be static.
                static int var = 3;
                           ^
1 error
```

Pour pouvoir utiliser une variable de classe dans une classe interne, il faut la déclarer dans sa classe englobante.

Il existe quatres types de classes internes :

- les classes internes non statiques : elles sont membres à part entière de la classe qui les englogent et peuvent accéder à tous les membres de cette dernière
- les classes internes locales : elles sont définies dans un block de code. Elles peuvent être static ou non.
- les classes internes anonymes : elles sont définies et instanciées à la volée sans posséder de nom
- les classe internes statiques : elles sont membres à part entière de la classe qui les englogent et peuvent accéder uniquement aux membres statiques de cette dernière

4.8.1. Les classes internes non statiques

Les classes internes non statiques (member inner−classes) sont définies dans une classe dite " principale " (top−level class) en tant que membre de cette classe. Leur avantage est de pouvoir accéder aux autres membres de la classe principale même ceux déclarés avec le modificateur private.

Exemple :

```
public class ClassePrincipale20 {
        private int valeur = 1;

        class ClasseInterne {
                public void afficherValeur() {
                        System.out.println("valeur = "+valeur);
                }
        }

        public static void main(String[] args) {
                ClassePrincipale20 cp = new ClassePrincipale20();
                ClasseInterne ci = cp. new ClasseInterne();
                ci.afficherValeur();
        }
}
```

Resultat :

```
C:\testinterne>javac ClassePrincipale20.java

C:\testinterne>java ClassePrincipale20
valeur = 1
```

Le mot clé this fait toujours référence à l'instance en cours. Ainsi this.var fait référence à la variable var de l'instance courante. L'utilisation du mot clé this dans une classe interne fait donc référence à l'instance courante de cette classe interne.

Exemple :

```
public class ClassePrincipale16 {
        class ClasseInterne {
                int var = 3;

                public void affiche() {
                        System.out.println("var      = "+var);
                        System.out.println("this.var = "+this.var);
                }
        }

        ClasseInterne ci = this. new ClasseInterne();

        public static void main(String[] args) {
                ClassePrincipale16 cp = new ClassePrincipale16();
                ClasseInterne ci = cp. new ClasseInterne();
                ci.affiche();
        }
}
```

Resultat :

```
C:\>java ClassePrincipale16
var      = 3
this.var = 3
```

Une classe interne a accès à tous les membres de sa classe principale. Dans le code, pour pouvoir faire référence à un membre de la classe principale, il suffit simplement d'utiliser son nom de variable.

Exemple :

```
public class ClassePrincipale17 {
        int valeur = 5;

        class ClasseInterne {
                int var = 3;

                public void affiche() {
                        System.out.println("var      = "+var);
                        System.out.println("this.var = "+this.var);
                        System.out.println("valeur   = "+valeur);
                }
        }

        ClasseInterne ci = this. new ClasseInterne();

        public static void main(String[] args) {
                ClassePrincipale17 cp = new ClassePrincipale17();
                ClasseInterne ci = cp. new ClasseInterne();
                ci.affiche();
        }
}
```

Resultat :

```
C:\testinterne>java ClassePrincipale17
var      = 3
this.var = 3
valeur   = 5
```

La situation se complique un peu plus, si la classe principale et la classe interne possède tous les deux un membre de même nom. Dans ce cas, il faut utiliser la version qualifée du mot clé this pour accéder au membre de la classe principale. La qualification se fait avec le nom de la classe principale ou plus généralement avec le nom qualifié d'une des classes englobantes.

Exemple :

```
public class ClassePrincipale18 {
        int var = 5;

        class ClasseInterne {
                int var = 3;

                public void affiche() {
                        System.out.println("var                          = "+var);
                        System.out.println("this.var                     = "+this.var);
                        System.out.println("ClassePrincipale18.this.var  = "
                          +ClassePrincipale18.this.var);
                }
        }

        ClasseInterne ci = this. new ClasseInterne();

        public static void main(String[] args) {
                ClassePrincipale18 cp = new ClassePrincipale18();
                ClasseInterne ci = cp. new ClasseInterne();
                ci.affiche();
        }
}
```

Resultat :

```
C:\>java ClassePrincipale18
var                          = 3
this.var                     = 3
ClassePrincipale18.this.var  = 5
```

Comme une classe interne ne peut être nommée du même nom que l'une de ces classes englobantes, ce nom qualifié est unique et il ne risque pas d'y avoir de confusion.

Le nom qualifié d'une classe interne est nom_classe_principale.nom_classe_interne. C'est donc le même principe que celui utilisé pour qualifié une classe contenue dans un package. La notation avec le point est donc légèrement étendue.

L'accès au membre de la classe principale est possible car le compilateur modifie le code de la classe principale et celui de la classe interne pour fournir à la classe interne une référence sur la classe principale.

Le code de la classe interne est modifié pour :

- ajouter une variable privée finale du type de la classe principale nommée this$0
- ajouter un paramètre supplémentaire dans le constructeur qui sera la classe principale et qui va initialiser la variable this$0
- utiliser cette variable pour préfixer les attributs de la classe principale utilisés dans la classe interne.

La code de la classe principale est modifié pour :

- ajouter une méthode static pour chaque champ de la classe principale qui attend en paramètre un objet de la classe principale. Cette méthode renvoie simplement la valeur du champ. Le nom de cette méthode est de la forme access$0
- modifier le code d'instanciation de la classe interne pour appeler le constructeur modifié

Dans le byte code généré, une variable privée finale contient une référence vers la classe principale. Cette variable est nommée this$0. Comme elle est générée par le compilateur, cette variable n'est pas utilisable dans le code source. C'est à partir de cette référence que le compilateur peut modifier le code pour accéder aux membres de la classe principale.

Pour pouvoir avoir acces aux membres de la classe principale, le compilateur génère dans la classe principale des accesseurs sur ces membres. Ainsi, dans la classe interne, pour accéder à un membre de la classe principale, le compilateur appelle un de ces accesseurs en utilisant la référence stockée. Ces méthodes ont un nom de la forme access$numero_unique et sont bien sûre inutilisables dans le code source puisqu'elles sont générées par le compilateur.

En tant que membre de la classe principale, une classe interne peut être déclarée avec le modificateur private ou protected.

Une classe peut faire référence dans le code source à son unique instance lors de l'éxécution via le mot clé this. Une classe interne possède au moins deux références :

- l'instance de la classe interne elle même
- éventuellement les instances des classes internes dans laquelle la classe interne est imbriquée
- l'instance de sa classe principale

Dans la classe interne, il est possible pour accéder à une de ces instances d'utiliser le mot clé this préfixé par le nom de la classe suivi d'un point :
nom_classe_principale.this
nom_classe_interne.this

Le mot this seul désigne toujours l'instance de la classe courante dans son code source, donc this seul dans une classe interne désigne l'instance de cette classe interne.

Une classe interne non statique doit toujours être instanciée relativement à un objet implicite ou explicite du type de la classe principale. A la compilation, le compilateur ajoute dans la classe interne une référence vers la classe principale contenu dans une variable privée nommée this$0. Cette référence est initialisée avec un paramètre fourni au constructeur de la classe interne. Ce mécanisme permet de lier les deux instances.

La création d'une classe interne nécessite donc obligatoirement une instance de sa classe principale. Si cette instance n'est pas accessible, il faut en créer une et utiliser une notation particulière de l'opérateur new pour pouvoir instancier la classe interne. Par défaut, lors de l'instanciation d'une classe interne, si aucune instance de la classe principale n'est utilisée, c'est l'instance courante qui est utilisée (mot clé this).

Exemple :

```
public class ClassePrincipale14 {
        class ClasseInterne {
        }

        ClasseInterne ci = this. new ClasseInterne();
}
```

Pour créer une instance d'une classe interne dans une méthode statique de la classe principale, (la méthode main() par exemple), il faut obligatoirement instancier un objet de la classe principale avant et utiliser cet objet lors de la création de l'instance de la classe interne. Pour créer l'instance de la classe interne, il faut alors utiliser une syntaxe particulière de l'opérateur new.

Exemple :

```
public class ClassePrincipale15 {
        class ClasseInterne {
        }
```

```
        ClasseInterne ci = this. new ClasseInterne();

        static void maMethode() {
                ClassePrincipale15 cp = new ClassePrincipale15();
                ClasseInterne ci = cp. new ClasseInterne();
        }
}
```

Il est possible d'utiliser une syntaxe condensée pour créer les deux instances en une seul et même ligne de code.

Exemple :

```
public class ClassePrincipale19 {
        class ClasseInterne {
        }

        static void maMethode() {
                ClasseInterne ci = new ClassePrincipale19(). new ClasseInterne();
        }
}
```

Une classe peut hériter d'une classe interne. Dans ce cas, il faut obligatoirement fournir aux contructeurs de la classe une référence sur la classe principale de la classe mère et appeler explicitement dans le constructeur le constructeur de cette classe principale avec une notation particulière du mot clé super

Exemple :

```
public class ClassePrincipale9 {
        public class ClasseInterne {
        }

        class ClasseFille extends ClassePrincipale9.ClasseInterne {
                ClasseFille(ClassePrincipale9 cp) {
                        cp. super();
                }
        }
}
```

Une classe interne peut être déclarée avec les modificateurs final et abstract. Avec le modificateur final, la classe interne ne pourra être utilisée comme classe mère. Avec le modificateur abstract, la classe interne devra être étendue pour pouvoir être instanciée.

4.1. Les classes internes locales

Ces classes internes locales (local inner−classes) sont définies à l'intérieure d'une méthode ou d'un bloc de code. Ces classes ne sont utilisables que dans le bloc de code où elles sont définies. Les classes internes locales ont toujours accès aux membres de la classe englobante.

Exemple :

```
public class ClassePrincipale21 {
   int varInstance = 1;

   public static void main(String args[]) {
     ClassePrincipale21 cp = new ClassePrincipale21();
     cp.maMethode();
   }
```

```
   public void maMethode() {

     class ClasseInterne {
       public void affiche() {
         System.out.println("varInstance = " + varInstance);
       }
     }

     ClasseInterne ci = new ClasseInterne();
     ci.affiche();
   }
 }
```

Resultat :

```
C:\testinterne>javac ClassePrincipale21.java

C:\testinterne>java ClassePrincipale21
varInstance = 1
```

Leur particularité, en plus d'avoir un accès aux membres de la classe principale, est d'avoir aussi un accès à certaines variables locales du bloc ou est définie la classe interne.

Ces variables définies dans la méthode (variables ou paramètres de la méthode) sont celles qui le sont avec le mot clé final. Ces variables doivent être initialisée avant leur utilisation par la classe interne. Celles ci sont utilisables n'importe ou dans le code de la classe interne.

Le modificateur final désigne une variable dont la valeur ne peut être changée une fois qu'elle a été initialisée.

Exemple :

```
public class ClassePrincipale12 {

   public static void main(String args[]) {
     ClassePrincipale12 cp = new ClassePrincipale12();
     cp.maMethode();
   }

   public void maMethode() {
        int varLocale = 3;

     class ClasseInterne {
       public void affiche() {
         System.out.println("varLocale = " + varLocale);
       }
     }

     ClasseInterne ci = new ClasseInterne();
     ci.affiche();
   }
 }
```

Resultat :

```
javac ClassePrincipale12.java
ClassePrincipale12.java:14: Attempt to use a non-final variable varLocale from a
 different method. From enclosing blocks, only final local variables are availab
le.
         System.out.println("varLocale = " + varLocale);
                                              ^
1 error
```

Cette restriction est imposée par la gestion du cycle de vie d'une variable locale. Une telle variable n'existe que durant l'éxécution de cette méthode. Une variable finale est une variable dont la valeur ne peut être modifiée après son initialisation. Ainsi, il est possible sans risque pour le compilateur d'ajouter un membre dans la classe interne et de copier le contenu de la variable finale dedans.

Exemple :

```
public class ClassePrincipale13 {

   public static void main(String args[]) {
     ClassePrincipale13 cp = new ClassePrincipale13();
     cp.maMethode();
   }

   public void maMethode() {
        final int varLocale = 3;

     class ClasseInterne {
       public void affiche(final int varParam) {
         System.out.println("varLocale = " + varLocale);
         System.out.println("varParam  = " + varParam);
       }
     }

     ClasseInterne ci = new ClasseInterne();
     ci.affiche(5);
   }
 }
```

Resultat :

```
C:\>javac ClassePrincipale13.java

C:\>java ClassePrincipale13
varLocale = 3
varParam  = 5
```

Pour permettre à une classe interne locale d'accéder à une variable locale utilisée dans le bloc de code ou est définie la classe interne, la variable doit être stockée dans un endroit ou la classe interne pourra y accéder. Pour que cela fonctionne, le compilateur ajoute les variables nécessaires dans le constructeur de la classe interne.

Les variables accédées sont dupliquées dans la classe interne par le compilateur. Il ajoute pour chaque variable un membre privé dans la classe interne dont le nom est de la forme val$nom_variable. Comme la variable accédée est déclarée finale, cette copie peut être faite sans risque. La valeur de chacune de ces variables est fournie en paramètre du constructeur qui a été modifié par le compilateur.

Une classe qui est définie dans un bloc de code n'est pas un membre de la classe englobante : elle n'est donc pas accessible en dehors du bloc de code ou elle est définie. Ces restrictions sont équivalentes à la déclaration d'une variable dans un bloc de code.

Les variables ajoutées par le compilateur sont préfixées par this$ et val$. Ces variables et le constructeur modifié par le compilateur ne sont pas utilisables dans le code source.

Etant visible uniquement dans le bloc de code qui la définie, une classe interne locale ne peut pas utiliser les modificateurs public, private, protected et static dans sa définition. Leur utilisation provoque une erreur à la compilation.

Exemple :

```
public class ClassePrincipale11 {
        public void maMethode() {
                public class ClasseInterne {
                }
        }
}
```

Resultat :

```
javac ClassePrincipale11.java
ClassePrincipale11.java:2: '}' expected.
        public void maMethode() {
                                 ^
ClassePrincipale11.java:3: Statement expected.
                public class ClasseInterne {
                ^
ClassePrincipale11.java:7: Class or interface declaration expected.
}
^
3 errors
```

4.8.3. Les classes internes anonymes

Les classes internes anonymes (anonymous inner−classes) sont des classes internes qui ne possèdent pas de nom. Elles ne peuvent donc être instanciées qu'à l'endroit ou elles sont définies.

Ce type de classe est très pratique lorsqu'une classe doit être utilisée une seule fois : c'est par exemple le cas d'une classe qui doit être utilisée comme un callback.

Une syntaxe particulière de l'opérateur new permet de déclarer et instancier une classe interne :
new classe_ou_interface () {
// définition des attributs et des méthodes de la classe interne
}

Cette syntaxe particulière utilise le mot clé new suivi d'un nom de classe ou interface que la classe interne va respectivement étendre ou implémenter. La définition de la classe suit entre deux accolades. Une classe interne anonyme peut soit hériter d'une classe soit implémenter une interface mais elle ne peut pas explicitement faire les deux.

Si la classe interne étend une classe, il est possible de fournir des paramètres entre les parenthèses qui suivent le nom de la classe. Ces arguments éventuels fournis au moment de l'utilisation de l'opérateur new sont passés au constructeur de la super classe. En effet, comme la classe ne possède pas de nom, elle ne possède pas non plus de constructeur.

Les classes internes anonymes qui implémentent une interface héritent obligatoirement de classe Object. Comme cette classe ne possèdent qu'un constructeur sans paramètre, il n'est pas possible lors de l'instanciation de la classe interne de lui fournir des paramètres.

Une classe interne anonyme ne peut pas avoir de constructeur puisqu'elle ne possède pas de nom mais elle peut avoir des initialisateurs.

Exemple :

```
public void init() {
  boutonQuitter.addActionListener(
    new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.exit(0);
      }
    }
  );
}
```

Les classes anonymes sont un moyen pratique de déclarer un objet sans avoir a lui trouver un nom. La contre partie est que cette classe ne pourra être instanciée dans le code qu'à l'endroit ou elle est définie : elle est déclarée et instanciée en un seul et unique endroit.

Le compilateur génère un fichier ayant pour nom la forme suivante : nom_classe_principale$numéro_unique. En fait, le compilateur attribut un numéro unique à chaque classe interne anonyme et c'est ce numéro qui est donné au nom du fichier préfixé par le nom de la classe englobante et d'un signe '$'.

4.8.4 Les classes internes statiques

Les classes internes statiques (static member inner−classes) sont des classes internes qui ne possèdent pas de référence vers leur classe principale. Elles ne peuvent donc pas accéder aux membres d'instance de leur classe englobante. Elles peuvent toutefois avoir accès aux variables statiques de la classe englobante.

Pour les déclarer, il suffit d'utiliser en plus le modificateur static dans la déclaration de la classe interne.

Leur utilisation est obligatoire si la classe est utilisée dans une méthode statique qui par définition peut être appelée dans avoir d'instance de la classe et que l'on ne peut pas avoir une instance de la classe englobante. Dans le cas contraire, le compilateur indiquera une erreur :

Exemple :

```
public class ClassePrincipale4 {

        class ClasseInterne {
                public void afficher() {
                        System.out.println("bonjour");
                }
        }

        public static void main(String[] args) {
                new ClasseInterne().afficher();
        }
}
```

Resultat :

```
javac ClassePrincipale4.java
ClassePrincipale4.java:10: No enclosing instance of class ClassePrincipale4 is i
n scope; an explicit one must be provided when creating inner class ClassePrinci
pale4. ClasseInterne, as in "outer. new Inner()" or "outer. super()".
                new ClasseInterne().afficher();
                ^
1 error
```

En déclarant la classe interne static, le code se compile et peut être exécuté

Exemple :

```
public class ClassePrincipale4 {

        static class ClasseInterne {
                public void afficher() {
                        System.out.println("bonjour");
                }
        }

        public static void main(String[] args) {
                new ClasseInterne().afficher();
        }
}
```

Resultat :

```
javac ClassePrincipale4.java
```

```
java ClassePrincipale4
bonjour
```

Comme elle ne possède pas de référence sur sa classe englobante, une classe interne statique est en fait traduite par le compilateur comme une classe principale. En fait, il est difficile de les mettre dans une catégorie (classe principale ou classe interne) car dans le code source c'est une classe interne (classe définie dans une autre) et dans le byte code généré c'est une classe principale.

Ce type de classe n'est pas très employée.

4.9. La gestion dynamique des objets

Tout objet appartient à une classe et Java sait la reconnaitre dynamiquement.

Java fournit dans son API un ensemble de classes qui permettent d'agir dynamiquement sur des classes. Cette technique est appellée introspection et permet :

- de décrire une classe ou une interface : obtenir son nom, sa classe mère, la liste de ces méthodes, de ses variables de classe, de ses constructeurs et de ses variables d'instances
- d'agir sur une classe en envoyant, à un objet Class des message comme à tout autre objet. Par exemple, créer dynamiquement à partir d'un objet Class une nouvelle instance de la classe représentée

Voir le chapitre sur l'introspection.

5. La bibliothèque de classes java

Dans la version 1.0, le JDK contient 8 packages chacun étant constitués par un ensemble de classes qui couvrent un même domaine et apportent de nombreuses fonctionnalités. Les différentes versions du JDK sont constamment enrichis de packetages :

Packages	JDK 1.0	JDK 1.1	JDK 1.2	JDK 1.3	Role
java.applet	X	X	X	X	classes pour développer des applets
java.awt	X	X	X	X	classes définissant un toolkit pour interfaces graphiques
java.awt.color			X	X	classes qui permettent de gérer et d'utiliser les couleurs
java.awt.datatransfer		X	X	X	classes qui permettent d'échanger des données via le presse−papier
java.awt.dnd			X	X	classes qui permettent de gérer cliquer/glisser
java.awt.event		X	X	X	classes qui permettent de gérer les évenements utilisateurs
java.awt.font			X	X	classes qui permettent d'utiliser les fontes
java.awt.geom			X	X	classes qui permettent de dessiner des formes géométiques
java.awt.im				X	
java.awt.im.spi				X	
java.awt.image		X	X	X	classes qui permettent d'afficher des images
java.awt.image.renderable			X	X	classes qui permettent de modifier le rendu des images
java.awt.print			X	X	classes qui permettent de réaliser des impressions
java.beans		X	X	X	classes qui permettent de développer des composants

					réutilisables
java.beans.beancontext			X	X	
java.io	X	X	X	X	classes pour gérer les flux
java.lang	X	X	X	X	classes de base du langage
java.lang.ref			X	X	
java.lang.reflect		X	X	X	classes pour la réflexion (introspection)
java.math		X	X	X	classes pour des opérations mathématiques
java.net	X	X	X	X	classes pour utiliser les fonctionnalités réseaux
java.rmi		X	X	X	classes pour le développement d'objets distribués
java.rmi.activation			X	X	
java.rmi.dgc		X	X	X	
java.rmi.registry		X	X	X	
java.rmi.server		X	X	X	classes pour gérer les objets serveurs de RMI
java.security		X	X	X	classes pour gérer la signature et la certification
java.security.acl		X	X	X	
java.security.cert		X	X	X	
java.security.interfaces		X	X	X	
java.security.spec			X	X	
java.sql		X	X	X	classes JDBC pour l'accès aux bases de données
java.text		X	X	X	classes pour formatter des objets en texte
java.util	X	X	X	X	classes définissant des utilitaires divers
java.util.jar			X	X	classes pour gérer les fichiers jar
java.util.zip		X	X	X	classes pour gérer les fichiers zip
javax.accessibility			X	X	
javax.naming				X	
javax.naming.directory				X	

javax.naming.event				X	
javax.naming.ldap				X	
javax.naming.spi				X	
javax.rmi				X	
javax.rmi.CORBA				X	
javax.sound.midi				X	
javax.sound.midi.spi				X	
javax.sound.sampled				X	
javax.sound.sampled.spi				X	
javax.swing			X	X	classes Swing pour développer des interfaces graphiques
javax.swing.border			X	X	classes pour gérer les bordures des composants Swing
javax.swing.colorchooser			X	X	composant pour sélectionner une couleur
javax.swing.event			X	X	classes pour la gestion des évenements utilisateur des composants Swing
javax.swing.filechooser			X	X	composant pour sélectionner un fichier
javax.swing.plaf			X	X	classes pour la gestion de l'aspect des composants Swing
javax.swing.plaf.basic			X	X	
javax.swing.plaf.metal			X	X	classes pour gérer l'aspect metal des composants Swing
javax.swing.plaf.multi			X	X	
javax.swing.table			X	X	
javax.swing.text			X	X	
javax.swing.text.html			X	X	
javax.swing.text.html.parser			X	X	
javax.swing.text.rtf			X	X	
javax.swing.tree			X	X	classes pour gérer un composant JTree
javax.swing.undo			X	X	classes pour gérer les annulations d'opérations d'édition

javax.transaction				X	
org.omg.CORBA			X	X	
org.omg.CORBA_2_3				X	
org.omg.CORBA_2_3.portable				X	
org.omg.CORBA.DynAnyPackage			X	X	
org.omg.CORBA.ORBPackage			X	X	
org.omg.CORBA.portable			X	X	
org.omg.CORBA.TypeCodePackage			X	X	
org.omg.CosNaming			X	X	
org.omg.CosNaming.NamingContextPackage			X	X	
org.omg.SendingContext				X	
org.omg.stub.java.rmi				X	

Ce chapitre contient les sections suivantes :

5.1. Présentation du package java.lang

Ce package de base contient les classes fondamentales tel que Object, Class, Math, System, String, StringBuffer, Thread, les wrapper etc ...

La classe Math est détaillée dans le chapitre "les fonctions mathématiques".

La classe Class est détaillée dans le chapitre "la gestion dynamique des objets et l'introspection".

La classe Thread est détaillée dans le chapitre "le multitache".

Il contient également plusieurs classes qui permettent de demander des actions au système d'exploitation sur laquelle la machine virtuelle tourne, par exemple ClassLoader, Runtime, SecurityManager.

Ce package est implicitement importé dans tous les fichiers sources par le compilateur.

5.1.1. La classe Object

C'est la super classe de toutes les classes Java : toutes ces méthodes sont donc héritées par toutes les classes.

5.1.1.1. La méthode getClass()

La méthode getClass() renvoie un objet de la classe Class qui représente la classe de l'objet.

Le code suivant permet de connaitre le nom de la classe de l'objet

Exemple :
`String nomClasse = monObject.getClass().getName();`

5.1.1.2. La méthode toString()

La méthode toString() de la classe Object renvoie le nom de la classe , suivi du séparateur @, lui même suivi par la valeur de hachage de l'objet.

5.1.1.3. La méthode equals()

La méthode equals() implémente une comparaison par défaut. Sa définition dans Object compare les références : donc obj1.equals(obj2) ne renverra true que si obj1 et obj2 désignent le même objet. Dans une sous classe de Object, pour laquelle on a besoin de pouvoir dire que deux objets distincts peuvent être égaux, il faut redéfinir la méthode equals héritée de Object.

5.1.1.4. La méthode finalize()

A l'inverse de nombreux langage orienté objet tel que le C++ ou Delphi, le programmeur Java n'a pas à se préoccuper de la destruction des objets qu'il instancie. Ceux−ci sont détruits et leur emplacement mémoire est récupéré par le ramasse miette de la machine virtuelle dès qu'il n'y a plus de référence sur l'objet.

La machine virtuelle garantit que toutes les ressources Java sont correctement libérées mais, quand un objet encapsule une ressource indépendante de Java (comme un fichier par exemple), il peut être préférable de s'assurer que la ressource sera libérée quand l'objet sera détruit. Pour cela, la classe Object définit la méthode protected finalize, qui est appelée quand le ramasse miettes doit récupérer l'emplacement de l'objet ou quand la machine virtuelle termine son exécution

Exemple :

```
import java.io.*;

public class AccesFichier {
   private FileWriter fichier;

   public AccesFichier(String s) {
      try {
      fichier = new FileWriter(s);
      }
      catch (IOException e) {
         System.out.println("Impossible d'ouvrir le fichier
```

5.1.1.5. La méthode clone()

Si x désigne un objet obj1, l'exécution de x.clone() renvoie un second objet obj2, qui est une copie de obj1 : si obj1 est ensuite modifié, obj2 n'est pas affecté par ce changement.

Par défaut, la méthode clone(), héritée de Object fait une copie variable par variable : elle offre donc un comportement acceptable pour de très nombreuses sous classe de Object. Cependant comme le processus de duplication peut être délicat à gérer pour certaines classes (par exemple des objets de la classe Container), l'héritage de clone ne suffit pas pour qu'une classe supporte le clonage.

Pour permettre le clonage d'une classe, il faut implémenter dans la classe l'interface Cloneable.

La première chose que fait la méthode clone() de la classe Object, quand elle est appelée, est de tester si la classe implémente Cloneable. Si ce n'est pas le cas, elle lève l'exception CloneNotSupportedException.

5.1.2. La classe String

Une chaine de caractères est contenue dans un objet de la classe String

On peut initialiser une variable String sans appeler explicitement un contructeur : le compilateur se charge de créer un objet.

Exemple : deux déclarations de chaines identiques.

```
String uneChaine = «bonjour»;
String uneChaine = new String(«bonjour»);
```

Les objets de cette classe ont la particularité d'être constants. Chaque traitement qui vise à transformer un objet de la classe est implémenté par une méthode qui laisse l'objet d'origine inchangé et renvoie un nouvel objet String contenant les modifications.

Exemple :

```
private String uneChaine;
void miseEnMajuscule(String chaine) {
    uneChaine = chaine.toUpperCase()
}
```

Il est ainsi possible d'enchainer plusieurs méthodes :

Exemple :

```
uneChaine = chaine.toUpperCase().trim();
```

L'opérateur + permet la concatenation de chaines de caractères.

La comparaison de deux chaine doit se faire via la méthode equals() qui compare les objets eux même et non l'opérateur == qui compare les réferences de ces objets :

Exemple :

```
String nom1 = new String("Bonjour");
String nom2 = new String("Bonjour");
System.out.println(nom1 == nom2); // affiche false
System.out.println( nom1.equals(nom2)); // affiche true
```

Cependant dans un soucis d'efficacité, le compilateur ne duplique pas 2 constantes chaines de caractères : il optimise l'espace mémoire utilisé en utilisant le même objet. Cependant, l'appel explicite du constructeur ordonne au compilateur de créer un nouvel objet.

Exemple :

```
String nom1 = «Bonjour»;
String nom2 = «Bonjour»;
String nom3 = new String(«Bonjour»);
System.out.println(nom1 == nom2); // affiche true
System.out.println(nom1 == nom3); // affiche false
```

La classe String possède de nombreuses méthodes dont voici les principales :

Méthodes la classe String	Role
charAt(int)	renvoie le nieme caractère de la chaine
compareTo(String)	compare la chaine avec l'argument
concat(String)	ajoute l'argument à la chaine et renvoie la nouvelle chaine
endsWith(String)	vérifie si la chaine se termine par l'argument
equalsIgnoreCase(String)	compare la chaine sans tenir compte de la casse
indexOf(String)	renvoie la position de début à laquelle l'argument est contenu dans la chaine
lastIndexOf(String)	renvoie la dernière position à laquelle l'argument est contenu dans la chaine
lenght()	renvoie la longueur de la chaine
replace(char,char)	renvoie la chaine dont les occurences d'un caractère ont remplacées
startsWidt(String int)	Vérifie si la chaine commence par la sous chaine
substring(int,int)	renvoie une partie de la chaine
toLowCase()	renvoie la chaine en minuscule
toUpperCase()	renvoie la chaine en majuscule
trim()	enlève les caractères non significatifs de la chaine

5.1.3. La classe StringBuffer

Les objets de cette classe contiennent des chaines de caractères variables, ce qui permet de les agrandir ou de les réduire. Cette objet peut être utilisé pour construire ou modifier une chaine de caractères chaque fois que l'utilisation de la classe String nécessiterait de nombreuses instanciations d'objets temporaires.

Par exemple, si str est un objet de type String, le compilateur utilisera la classe StringBuffer pour traiter la concaténation de « abcde »+str+« z » en générant le code suivant : new StringBuffer().append(« abcde ».append(str).append(« z »).toString();

Ce traitement aurait pu être réalisé avec trois appels à la méthode concat() de la classe String mais chacun des appels aurait instancié un objet StringBuffer pour réaliser la concaténation, ce qui est couteux en temps exécution

La classe StringBuffer dispose de nombreuses méthodes qui permettent de modifier le contenu de la chaine de caractère

Exemple (code java 1.1) : une méthode uniquement pédagogique

```
public class MettreMaj {

   static final String lMaj = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   static final String lMin = "abcdefghijklmnopqrstuvwxyz";

   public static void main(java.lang.String[] args) {
      System.out.println(MetMaj("chaine avec MAJ et des min"));
   }

   public static String MetMaj(String s) {
      StringBuffer sb = new StringBuffer(s);

      for ( int i = 0; i <sb.length(); i++) {
          int index = lMin.indexOf(sb.charAt(i));
          if (index >=0 ) sb.setCharAt(i,lMaj.charAt(index));
      }
      return sb.toString();
   }
}
```

Résultat :

```
CHAINE AVEC MAJ ET DES MIN
```

5.1.4. Les wrappers

Les objet de type wrappers (enveloppeurs) représentent des objets qui encapsulent une données de type primitif et qui fournissent un ensemble de méthodes qui permettent notamment de faire de conversions.

Ces classes offrent toutes les services suivants :

- un constructeur qui permet une instanciation à partir du type primitif et un constructeur qui permet une instanciation à partir d'un objet String
- une méthode pour fournir la valeur primitive représentée par l'objet
- une méthode equals() pour la comparaison.

Les méthodes de conversion opèrent sur des instances, mais il est possible d'utiliser des méthodes statiques.

Exemple :

```
int valeur =
Integer.valueOf("999").intValue();
```

Ces classes ne sont pas interchangeable avec les types primitif d'origine car il s'agit d'objet.

Exemple :

```
Float objetpi = new Float(«3.1415»);

System.out.println(5*objetpi); // erreur à la compil
```

Pour obtenir la valeur contenue dans l'objet, il faut utiliser la méthode typeValue() ou type est le nom du type standard

Exemple :

```
Integer Entier = new Integer(«10»);

int entier = Entier.intValue();
```

Les classes Integer, Long, Float et Double définissent toutes les constantes MAX_VALUE et MIN_VALUE qui représentent leurs valeurs minimales et maximales.

Lorsque l'on effectue certaines opérations mathématiques sur des nombres à virgules flottantes (float ou double), le résultat peut prendre l'une des valeurs suivantes :

- NEGATIVE_INFINITY : infini négatif causé par la division d'un nombre négatif par 0.0
- POSITIVE_INFINITY : infini positif causé par la division d'un nombre positif par 0.0
- NaN: n'est pas un nombre (Not a Number) causé par la division de 0.0 par 0.0

Il existe des méthodes pour tester le résultat :

Float.isNaN(float); //idem avec double
Double.isInfinite(double); // idem avec float

Exemple :

```
float res = 5.0f / 0.0f;

if (Float.isInfinite(res)) { … };
```

La constante Float.NaN n'est ni égale à un nombre dont la valeur est NaN ni à elle même. Float.NaN == Float.NaN retourne False

Lors de la division par zéro d'un nombre entier, une exception est levée.

Exemple :

```
System.out.println(10/0);

Exception in thread "main" java.lang.ArithmeticException: / by zero
at test9.main(test9.java:6)
```

5.1.5. La classe System

Cette classe possède de nombreuses fonctionnalités pour utiliser des services du système d'exploitation.

5.1.5.1. L'utilisation des flux d'entrée/sortie standard

La classe System défini trois variables statiques qui permettent d'utiliser les flux d'entrée/sortie standards du système d'exploitation.

Variable	Type	Rôle
in	PrintStream	Entrée standard du système. Par défaut, c'est le clavier.
out	InputStream	Sortie standard du système. Par défaut, c'est le moniteur.
err	PrintStream	Sortie standard des erreurs du système. Par défaut, c'est le moniteur.

Exemple :

```
System.out.println("bonjour");
```

La classe système possède trois méthodes qui permettent de rediriger ces flux.

5.1.5.2. Les variables d'environnement et les propriétés du système

JDK 1.0 propose la méthode statique getEnv() qui renvoie la valeur de la propriété système dont le nom est fourni en paramètre.

Depuis le JDK 1.1, cette méthode est deprecated car elle n'est pas multi−système. Elle est remplacée par un autre mécanisme qui n'interroge pas directement le système mais qui recherche les valeurs dans un ensemble de propriétés. Cet ensemble est consituté de propriétés standard fournies par l'environnement java et par des propriétés ajoutés par l'utilisateur.

Voici une liste non exhaustive des propriétés fournies par l'environnement java :

Nom de la propriété	Rôle
java.version	Version du JRE
java.vendor	Auteur du JRE
java.vendor.url	URL de l'auteur
java.home	Répertoire d'installation de java
java.vm.version	Version de l'implémentation la JVM
java.vm.vendor	Auteur de l'implémentation de la JVM
java.vm.name	Nom de l'implémentation de la JVM
java.specification.version	Version des spécifications de la JVM
java.specification.vendor	Auteur des spécifications de la JVM
java.specification.name	Nom des spécifications de la JVM
java.ext.dirs	Chemin du ou des répertoires d'extension
os.name	Nom du système d'exploitation
os.arch	Architecture du système d'exploitation

os.version	Version du système d'exploitation
file.separator	Séparateur de fichiers (exemple : "/" sous Unix, "\" sous Windows)
path.separator	Séparateur de chemin (exemple : ":" sous Unix, ";" sous Windows)
line.separator	Séparateur de ligne
user.name	Nom du user courant
user.home	Répertoire d'accueil du user courant
user.dir	Répertoire courant au moment de l'initialisation de la propriété

Pour définir ces propres propriétés, il faut utiliser l'option −D de l'interpreteur java en utilisant la ligne de commande.

La méthode statique getProperty() permet d'obtenir la valeur de la propriété dont le nom est fourni en paramètre. Une version surchargée de cette méthode permet de préciser un second paramètre qui contiendra la valeur par défaut, si la propriété n'est pas définie.

Exemple :

```
public class TestProperty {

  public static void main(String[] args) {
    System.out.println("java.version                 ="+System.getProperty("java.version"));
    System.out.println("java.vendor                  ="+System.getProperty("java.vendor"));
    System.out.println("java.vendor.url              ="+System.getProperty("java.vendor.url"));
    System.out.println("java.home                    ="+System.getProperty("java.home"));
    System.out.println("java.vm.specification.version ="
      +System.getProperty("java.vm.specification.version"));
    System.out.println("java.vm.specification.vendor  ="
      +System.getProperty("java.vm.specification.vendor"));
    System.out.println("java.vm.specification.name    ="
      +System.getProperty("java.vm.specification.name"));
    System.out.println("java.vm.version              ="+System.getProperty("java.vm.version"));
    System.out.println("java.vm.vendor               ="+System.getProperty("java.vm.vendor"));
    System.out.println("java.vm.name                 ="+System.getProperty("java.vm.name"));
    System.out.println("java.specification.version    ="
      +System.getProperty("java.specification.version"));
    System.out.println("java.specification.vendor     ="
      +System.getProperty("java.specification.vendor"));
    System.out.println("java.specification.name       ="
      +System.getProperty("java.specification.name"));
    System.out.println("java.class.version            ="
      +System.getProperty("java.class.version"));
    System.out.println("java.class.path               ="
      +System.getProperty("java.class.path"));
    System.out.println("java.ext.dirs                ="+System.getProperty("java.ext.dirs"));
    System.out.println("os.name                      ="+System.getProperty("os.name"));
    System.out.println("os.arch                      ="+System.getProperty("os.arch"));
    System.out.println("os.version                   ="+System.getProperty("os.version"));
    System.out.println("file.separator               ="+System.getProperty("file.separator"));
    System.out.println("path.separator               ="+System.getProperty("path.separator"));
    System.out.println("line.separator               ="+System.getProperty("line.separator"));
    System.out.println("user.name                    ="+System.getProperty("user.name"));
    System.out.println("user.home                    ="+System.getProperty("user.home"));
    System.out.println("user.dir                     ="+System.getProperty("user.dir"));
  }
}
```

Par défaut, l'accès aux propriétés système est restreint par le SecurityManager pour les applets.

5.1.6. La classe Runtime

La classe Runtime permet d'intéragir avec le système dans lequel l'application s'éxécute : obtenir des informations sur le système, arrêt de la machine virtuelle, éxecution d'un programme externe

Cette classe ne peut pas être instanciée mais il est possible d'obtenir une instance en appelant la méthode statique getRuntime().

La méthode exec() permet d'éxécuter des commandes du système d'exploitation ou s'éxécute la JVM. Elle lance la commande de manière asynchrone et renvoie un objet de type Process pour obtenir des informations sur le processus lancé.

L'inconvénient d'utiliser cette méthode est que la commande executée est dépendante du système d'exploitation.

Cette section est en cours d'écriture

5.2. Présentation rapide du package awt java

AWT est une collection de classes pour la réalisation d'applications graphique ou GUI (Graphic User Interface)

Les composants qui sont utilisés par les classes définies dans ce package sont des composants dit "lourds" : ils dépendent entièrement du système d'explotation. D'ailleurs leur nombre est limité car ils sont communs à plusieurs systèmes d'exploitation pour assurer la portabilité. Cependant, la représentation d'une interface graphique avec awt sur plusieurs systèmes peut ne pas être identique.

La chapitre "Création d'interface graphique avec AWT" détaille l'utilisation de ce package,

5.2.1. Le package java.image

Ce package permet la gestion des images

5.2.2. Le package java.awt.perr

Ce package contient des classes qui réalise l'interface avec le système d'exploitation.

5.3. Présentation rapide du package java.io

Ce package définit un ensemble de classes pour la gestion des flux d'entrées−sorties

Le chapitre les flux détaille l'utilisation de ce package

5.4. Le package java.util

Ce package contient un ensemble de classes utilitaires : la gestion des dates (Date et Calendar), la génération de nombres aléatoires (Random), la gestion des collections ordonnées ou non tel que la tablea de hachage (HashTable), le vecteur (Vector), la pile (Stack) ..., la gestion des propriétés (Properties), des classes dédiées à l'internationalisation (ResourceBundle, PropertyResourceBundle, ListResourceBundle) etc ...

Certaines de ces classes sont présentées plus en détail dans les sections suivantes.

5.4.1. La classe StringTokenizer

Cette classe permet de découper une chaine de caractères (objet de type String) en fonction de séparateurs. Le constructeur de la classe accepte 2 paramètres : la chaine à décomposer et une chaine contenant les séparateurs

Exemple (code java 1.1) :

```
import java.util.*;

class test9 {
   public static void main(String args[]) {
      StringTokenizer st = new StringTokenizer("chaine1,chaine2,chaine3,chaine4",",");
      while (st.hasMoreTokens()) {
         System.out.println((st.nextToken()).toString());
      }
   }
}

C:\java>java test9
chaine1
chaine2
chaine3
chaine4
```

La méthode hasMoreTokens() fournit un contrôle d'itération sur la collection en renvoyant un booleen indiquant si il reste encore des éléments.

La méthode getNextTokens() renvoie le prochain élément sous la forme d'un objet String

5.4.2. La classe Random

La classe Random permet de générer des nombres pseudo−aléatoires. Après l'appel au constructeur, il suffit d'appeler la méthode correspondant au type désiré : nextInt(), nextLong(), nextFloat() ou nextDouble()

Méthodes	valeur de retour
nextInt	entre Integer.MIN_VALUE et Interger.MAX_VALUE
nextLong	entre long.MIN_VALUE et long.MAX_VALUE
nextFloat ou nextDouble	entre 0.0 et 1.0

Exemple (code java 1.1) :

```
import java.util.*;
class test9 {
   public static void main (String args[]) {
      Random r = new Random();
      int a = r.nextInt() %10; //entier entre −9 et 9
      System.out.println("a = "+a);
   }
}
```

5.4.3. Les classes Date et Calendar

En java 1.0, la classe Date permet de manipuler les dates.

Exemple (code java 1.0) :

```
import java.util.*;
…
    Date maintenant = new Date();
    if (maintenant.getDay() == 1)
       System.out.println(" lundi ");
```

Le constructeur d'un objet Date l'initialise avec la date et l'heure courante du système.

Exemple (code java 1.0) : recherche et affichage de l'heure

```
import java.util.*;
import java.text.*;

public class TestHeure {
   public static void main(java.lang.String[] args) {
      Date date = new Date();
      System.out.println(DateFormat.getTimeInstance().format(date));
   }
}
```

Résultat :

```
22:05:21
```

La méthode getTime() permet de calculer le nombre de millisecondes écoulées entre la date qui est encapsulée dans l'objet qui reçoit le message getTime et le premier janvier 1970 à minuit GMT.

En java 1.1, de nombreuses méthodes et constructeurs de la classe Date sont deprecated, notamment celles qui permettent de manipuler les éléments qui composent la date et leur formattage : il faut utiliser la classe Calendar.

Exemple (code java 1.1) :

```
import java.util.*;

public class TestCalendar {
   public static void main(java.lang.String[] args) {

      Calendar c = Calendar.getInstance();
      if (c.get(Calendar.DAY_OF_WEEK) == Calendar.MONDAY)
          System.out.println(" nous sommes lundi ");
      if (c.get(Calendar.DAY_OF_WEEK) == Calendar.TUESDAY)
          System.out.println(" nous sommes mardi ");
      if (c.get(Calendar.DAY_OF_WEEK) == Calendar.WEDNESDAY)
          System.out.println(" nous sommes mercredi ");
      if (c.get(Calendar.DAY_OF_WEEK) == Calendar.THURSDAY)
          System.out.println(" nous sommes jeudi ");
      if (c.get(Calendar.DAY_OF_WEEK) == Calendar.FRIDAY)
          System.out.println(" nous sommes vendrei ");
      if (c.get(Calendar.DAY_OF_WEEK) == Calendar.SATURDAY)
          System.out.println(" nous sommes samedi ");
      if (c.get(Calendar.DAY_OF_WEEK) == Calendar.SUNDAY)
          System.out.println(" nous sommes dimanche ");
```

```
   }
}
```

Résultat :

```
 nous sommes lundi
```

5.4.4. La classe Vector

Un objet de la classe Vector peut être considéré comme une tableau évolué qui peut contenir un nombre indéterminé d'objets.

Les méthodes principales sont les suivantes :

Méthode	Rôle
void addElement(Object)	ajouter un objet dans le vecteur
boolean contains(Object)	retourne true si l'objet est dans le vecteur
Object elementAt(int)	retourne l'objet à l'index indiqué
Enumeration elements()	retourne une enumeration contenant tous les éléments du vecteur
Object firstElement()	retourne le premier élément du vecteur (celui dont l'index est égal à zéro)
int indexOf(Object)	renvoie le rang de l'élément ou −1
void insertElementAt(Object, int)	insérer un objet à l'index indiqué
boolean isEmpty()	retourne un booléen si le vecteur est vide
Objet lastElement()	retourne le dernier élément du vecteur
void removeAllElements()	vider le vecteur
void removeElement(Object)	supprime l'objet du vecteur
void removeElementAt(int)	supprime l'objet à l'index indiqué
void setElementAt(object, int)	remplacer l'élément à l'index par l'objet
int size()	nombre d'objet du vecteur

On peut stocker des objets de classes différentes dans un vecteur mais les éléments stockés doivent obligatoirement être des objets (pour le type primitif il faut utiliser les wrappers tel que Integer ou Float mais pas int ou float).

Exemple (code java 1.1) :

```
Vector v = new Vector();
v.addElement(new Integer(10));
v.addElement(new Float(3.1416));
v.insertElementAt("chaine ",1);
System.out.println(" le vecteur contient "+v.size()+ " elements ");
String retrouve = (String) v.elementAt(1);
System.out.println(" le 1er element = "+retrouve);

C:\$user\java>java test9
```

```
le vecteur contient 3 elements
le 1er element = chaine
```

Exemple (code java 1.1) : le parcours d'un vecteur

```
   Vector v = new Vector();
   ...

   for (int i = 0; i < v.size() ; i ++) {
      System.out.println(v.elementAt(i));
   }
```

Pour un parcours en utilisant l'interface Enumeration, voir le chapitre correspondant.

5.4.5. La classe Hashtable

Les informations d'une Hastable sont stockées sous la forme clé − données. Cet objet peut être considéré comme un dictionnaire.

Exemple (code java 1.1) :

```
Hashtable dico = new Hashtable();
dico.put(«livre1», « titre du livre 1 »);
dico.put(«livre2», «titre du livre 2 »);
```

Il est possible d'utiliser n'importe quel objet comme clé et comme donnée

Exemple (code java 1.1) :

```
dico.put(«jour», new Date());
dico.put(new Integer(1),«premier»);
dico.put(new Integer(2),«deuxième»);
```

Pour lire dans la table, on utilise get(object) en donnant la clé en paramètre.

Exemple (code java 1.1) :

```
System.out.println(« nous sommes le » +dico.get(«jour»));
```

La méthode remove(Object) permet de supprimer une entrée du dictionnaire correspondant à la clé passée en paramètre.

La méthode size() permet de connaitre le nombre d'association du dictionnaire.

5.4.6. L'interface Enumeration

L'interface Enumeration est utilisée pour permettre le parcours séquentiel de collections.

Enumeration est une interface qui définit 2 méthodes :

Méthodes	Rôle

boolean hasMoreElements()	retourne true si l'énumeration contient encore un ou plusieurs elements
Object nextElement()	retourne l'objet suivant de l'énumération Elle leve une Exception NoSuchElementException si la fin de la collection est atteinte.

Exemple (code java 1.1) : contenu d'un vecteur et liste des clés d'une Hastable

```
import java.util.*;

class test9 {

   public static void main (String args[]) {

      Hashtable h = new Hashtable();
      Vector v = new Vector();

      v.add("chaine 1");
      v.add("chaine 2");
      v.add("chaine 3");

      h.put("jour", new Date());
      h.put(new Integer(1),"premier");
      h.put(new Integer(2),"deuxième");

      System.out.println("Contenu du vector");

      for (Enumeration e = v.elements() ; e.hasMoreElements() ; ) {
         System.out.println(e.nextElement());
      }

      System.out.println("\nContenu de la hashtable");

      for (Enumeration e = h.keys() ; e.hasMoreElements() ; ) {
         System.out.println(e.nextElement());
      }
   }
}

C:\$user\java>java test9
Contenu du vector
chaine 1
chaine 2
chaine 3
Contenu de la hashtable
jour
2
1
```

5.4.7. Les expressions régulières

Le JDK 1.4 propose une api en standard pour utiliser les expressions régulières. Les expressions régulières permettent de comparer une chaine de caractères àun motif pour vérifier qu'il y a concordance.

La package java.util.regexp contient deux classes et une exception pour gérer les expressions régulières :

Classe	Rôle
Matcher	comparer une chaine de caractère avec un motif
Pattern	encapsule une version compilée d'un motif
PatternSyntaxException	exception levée lorsque le motif contient une erreur de syntaxe

5.4.7.1. Les motifs

Les expressions régulières utilisent un motif. Ce motif est une chaine de caractères qui contient des caractères et des métacaractères. Les métacaractères ont une signification particulière et sont interprétés.

Il est possible de déspécialiser un métacaractère (lui enlever sa signification particulière à en le faisant précédé d'un caractère backslash. Ainsi pour utiliser le caractère backslash, il faut le doubler.

Les métacaractères reconnus par l'api sont :

métacaractères	rôle
()	
[]	définir un ensemble de caractères
{}	définir une répétition du motif précédent
\	déspécalisation du caractère qui suit
^	debut de la ligne
$	fin de la ligne
|	le motif precedent ou le motif suivant
?	motif precédent répété zero ou une fois
*	motif précédent repeté zéro ou plusieurs fois
+	motif précédent répété une ou plusieurs fois
.	un caractère quelconque

Certains caractères spéciaux ont une notation particulière :

Notation	Rôle
\t	tabulation
\n	nouvelle ligne (ligne feed)
\\	backslash

Il est possible de définir des ensembles de caractères à l'aide des caractères [et]. Il suffit d'indiquer les caractères de l'ensemble entre ces deux caractères

Exemple : toutes les voyelles
`[aeiouy]`

Il est possible d'utiliser une plage de caractères consécutifs en séparant le caractère de début de la plage et le caractère de fin de la plage avec un caractère –

Exemple : toutes les lettres minuscules

```
[a-z]
```

L'ensemble peut être l'union de plusieurs plages.

Exemple : toutes les lettres
`[a-zA-Z]`

Par défaut l'ensemble [] désigne tous les caractères. Il est possible de définir un ensemble de la forme tous sauf ceux précisés en utilisant le caractère ^ suivi des caractères à enlever de l'ensemble

Exemple : tous les caractères sauf les lettres
`[^a-zA-Z]`

Il existe plusieurs ensembles de caractères prédéfinis :

Notation	Contenu de l'ensemble
\d	un chiffre
\D	tous sauf un chiffre
\w	une lettre ou un underscore
\W	tous sauf une lettre ou un underscore
\s	un séparateur (espace, tabulation, retour chariot, …)
\S	tous sauf un séparateur

Plusieurs métacaractères permettent de préciser un critère de répétition d'un motif

métacaractères	rôle
{n}	répétition du motif précédent n fois
{n,m}	répétition du motif précédent entre n et m fois
{n,}	répétition du motif précédent
?	motif precédent répété zero ou une fois
*	motif précédent repété zéro ou plusieurs fois
+	motif précédent répété une ou plusieurs fois

Exemple : la chaine AAAAA
`A{5}`

5.4.7.2. La classe Pattern

Cette classe encapsule une representation compilée d'un motif d'une expression régulière.

La classe Pattern ne possède pas de constructeur public mais propose une méthode statique compile().

Exemple :

```
private static Pattern motif = null;
...
motif = Pattern.compile("liste[0-9]");
```

Une version surchargée de la méthode compile() permet de préciser certaines options dont la plus interressante permet de rendre insensible à la casse les traitements en utilisant le flag CASE_INSENSITIVE.

Exemple :

```
private static Pattern motif = null;
...
motif = Pattern.compile("liste[0-9]",Pattern.CASE_INSENSITIVE);
```

Cette méthode compile() renvoie une instance de la classe Pattern si le motif est syntaxiquement correcte sinon elle lève une exception de type PatternSyntaxException.

La méthode matches(String, String) permet de rapidement et facilement utiliser les expressions régulières avec un seul appel de méthode en fournissant le motif est la chaine à traiter.

Exemple :

```
if (Pattern.matches("liste[0-9]","liste2")) {
      System.out.println("liste2 ok");
} else {
      System.out.println("liste2 ko");
}
```

5.4.7.3. La classe Matcher

La classe Matcher est utilisée pour effectuer la comparaison entre une chaine de caractères et un motif encapsulé dans un objet de type Pattern.

Cette classe ne possède aucun constructeur public. Pour obtenir une instance de cette classe, il faut utiliser la méthode matcher() d'une instance d'un objet Pattern en lui fournissant la chaine à traiter en paramètre.

Exemple :

```
motif = Pattern.compile("liste[0-9]");
matcher = motif.matcher("liste1");
```

La méthodes matches() tente de comparer toute la chaine avec le motif et renvoie le résultat de cette comparaison.

Exemple :

```
motif = Pattern.compile("liste[0-9]");
```

```
matcher = motif.matcher("liste1");
if (matcher.matches()) {
      System.out.println("liste1 ok");
} else {
      System.out.println("liste1 ko");
}
matcher = motif.matcher("liste10");
if (matcher.matches()) {
      System.out.println("liste10 ok");
} else {
      System.out.println("liste10 ko");
}
```

Résultat :

```
liste1 ok
liste10 ko
```

La méthode lookingAt() tente de recherche le motif dans la chaine à traiter

Exemple :

```
motif = Pattern.compile("liste[0-9]");
matcher = motif.matcher("liste1");
if (matcher.lookingAt()) {
      System.out.println("liste1 ok");
} else {
      System.out.println("liste1 ko");
}
matcher = motif.matcher("liste10");
if (matcher.lookingAt()) {
      System.out.println("liste10 ok");
} else {
      System.out.println("liste10 ko");
}
```

Résultat :

```
liste1 ok
liste10 ok
```

La méthode find() permet d'obtenir des informations sur chaque occurrence ou le motif est trouvé dans la chaine à traiter.

Exemple :

```
matcher = motif.matcher("zzliste1zz");
if (matcher.find()) {
      System.out.println("zzliste1zz ok");
} else {
     System.out.println("zzliste1zz ko");
}
```

Résultat :

```
zzliste1zz ok
```

Il est possible d'appeler successivement cette méthode pour obtenir chacune des occurrences.

Exemple :

```
        int i = 0;
        motif = Pattern.compile("liste[0-9]");
        matcher = motif.matcher("liste1liste2liste3");
        while (matcher.find()) {
              i++;
        }
        System.out.println("nb occurences = " + i);
```

Les méthodes start() et end() permettent de connaitre la position de début et de fin dans la chaine dans l'occurrence en cours de traitement.

Exemple :

```
        int i = 0;
        motif = Pattern.compile("liste[0-9]");
        matcher = motif.matcher("liste1liste2liste3");
        while (matcher.find()) {
              i++;
        }
        System.out.println("nb occurences = " + i);
```

Résultat :

```
  pos debut : 0 pos fin : 6
  pos debut : 6 pos fin : 12
  pos debut : 12 pos fin : 18
  nb occurences = 3
```

La classe Matcher propose aussi les méthodes replaceFirst() et replaceAll() pour facilement remplacer la première ou toutes les occurrences du motif trouvées par une chaine de caractères.

Exemple : remplacement de la première occurrence

```
        motif = Pattern.compile("liste[0-9]");
        matcher = motif.matcher("zz liste1 zz liste2 zz");
        System.out.println(matcher.replaceFirst("chaine"));
```

Résultat :

```
zz chaine zz liste2 zz
```

Exemple : remplacement de toutes les occurrences

```
      motif = Pattern.compile("liste[0-9]");
      matcher = motif.matcher("zz liste1 zz liste2 zz");
      System.out.println(matcher.replaceAll("chaine"));
```

Résultat :

```
zz chaine zz chaine zz
```

5.5. Présentation rapide du package java.net

Ce package contient un ensemble de classes pour permettre une interaction avec le réseau

Le chapitre l'interaction avec le reseau détaille l'utilisation de ce package

5.6. Présentation rapide du package java.applet

Ce package contient des classes à importer obligatoirement pour développer des applets

Le développement des applets est détaillé dans le chapitre les applets

6. Les fonctions mathématiques

La classe java.lang.Math contient une série de méthodes et variables mathématiques. Comme la classe Math fait partie du package java.lang, elle est automatiquement importée. Il n'est pas nécessaire de déclarer un objet de type Math car les méthodes sont toutes static

Exemple (code java 1.1) : Calculer et afficher la racine carrée de 3

```
public class Math1 {
   public static void main(java.lang.String[] args) {
      System.out.println(" = " + Math.sqrt(3.0));
   }
}
```

6.1. Les variables de classe

PI représente pi dans le type double (3,14159265358979323846)

E représente e dans le type double (2,7182818284590452354)

Exemple (code java 1.1) :

```
public class Math2 {
        public static void main(java.lang.String[] args) {
        System.out.println(" PI = "+Math.PI);
                System.out.println(" E = "+Math.E);
        }
}
```

6.2. Les fonctions trigonométriques

Les méthodes sin(), cos(), tan(), asin(), acos(), atan() sont déclarées : public static double fonctiontrigo(double a)

Les angles doivent être exprimés en radians. Pour convertir des degrés en radian, il suffit du multiplier par PI/180

6.3. Les fonctions de comparaisons

max (n1, n2)
min (n1, n2)

Ces méthodes existent pour les types int, long, float et double : elles déterminent respectivement les valeurs maximales et minimales des deux paramètres.

Exemple (code java 1.1) :

```
public class Math1 {
        public static void main(String[] args) {
                System.out.println(" le plus grand = " + Math.max(5, 10));
                System.out.println(" le plus petit = " + Math.min(7, 14));
        }
}
```

Résultat :

```
le plus grand = 10
le plus petit = 7
```

6.4. Les arrondis

6.4.1. La méthode round(n)

Cette méthode ajoute 0,5 à l'argument et restitue la plus grande valeur entière (int) inférieure ou égale au résultat. La méthode est définie pour les types float et double.

Exemple (code java 1.1) :

```
public class Math1 {
        public static void main(String[] args) {
        System.out.println(" round(5.0) = "+Math.round(5.0));
        System.out.println(" round(5.2) = "+Math.round(5.2));
        System.out.println(" round(5.5) = "+Math.round(5.5));
        System.out.println(" round(5.7) = "+Math.round(5.7));
        }
}
```

Résultat :

```
round(5.0) = 5
round(5.2) = 5
round(5.5) = 6
round(5.7) = 6
```

6.4.2. La méthode rint(double)

Cette méthode effectue la même opération mais renvoie un type double

Exemple (code java 1.1) :

```
public class Math1 {
        public static void main(String[] args) {
        System.out.println(" rint(5.0) = "+Math.rint(5.0));
        System.out.println(" rint(5.2) = "+Math.rint(5.2));
        System.out.println(" rint(5.5) = "+Math.rint(5.5));
        System.out.println(" rint(5.7) = "+Math.rint(5.7));
        }
}
```

Résultat :

```
rint(5.0) = 5.0
rint(5.2) = 5.0
rint(5.5) = 6.0
rint(5.7) = 6.0
```

6.4.3. La méthode floor(double)

Cette méthode renvoie l'entier le plus proche inférieur ou égal à l'argument

Exemple (code java 1.1) :

```
public class Math1 {
        public static void main(String[] args) {
        System.out.println(" floor(5.0) = "+Math.floor(5.0));
        System.out.println(" floor(5.2) = "+Math.floor(5.2));
        System.out.println(" floor(5.5) = "+Math.floor(5.5));
        System.out.println(" floor(5.7) = "+Math.floor(5.7));
        }
}
```

résultat :

```
floor(5.0) = 5.0
floor(5.2) = 5.0
floor(5.5) = 5.0
floor(5.7) = 5.0
```

6.4.4. La méthode ceil(double)

Cette méthode renvoie l'entier le plus proche supérieur ou égal à l'argument

Exemple (code java 1.1) :

```
public class Math1 {
        public static void main(String[] args) {
        System.out.println(" ceil(5.0) = "+Math.ceil(5.0));
        System.out.println(" ceil(5.2) = "+Math.ceil(5.2));
        System.out.println(" ceil(5.5) = "+Math.ceil(5.5));
        System.out.println(" ceil(5.7) = "+Math.ceil(5.7));
        }
}
```

résultat :

```
ceil(5.0) = 5.0
ceil(5.2) = 6.0
ceil(5.5) = 6.0
ceil(5.7) = 6.0
```

6.4.5. La méthode abs(x)

Cette méthode donne la valeur absolue de x (les nombre négatifs sont convertis en leur opposé). La méthode est définie pour les types int, long, float et double.

Exemple (code java 1.1) :

```
public class Math1 {
        public static void main(String[] args) {
        System.out.println(" abs(−5.7) = "+abs(−5.7));
        }
}
```

Résultat :

```
abs(−5.7) = 5.7
```

6.4.6. La méthode IEEEremainder(double, double)

Cette méthode renvoie le reste de la division du premier argument par le deuxieme

Exemple (code java 1.1) :

```
public class Math1 {
        public static void main(String[] args) {
        System.out.println(" reste de la division de 3 par 10 = "
           +Math.IEEEremainder(10.0, 3.0) );
        }
}
```

Résultat :

```
reste de la division de 3 par 10 = 1.0
```

6.5. Les Exponentielles et puissances

6.5.1. La méthode pow(double, double)

Cette méthode elève le premier argument à la puissance indiquée par le second.

Exemple (code java 1.1) :

```
public static void main(java.lang.String[] args) {
        System.out.println(" 5 au cube  = "+Math.pow(5.0, 3.0) );
}
```

Résultat :

```
5 au cube  = 125.0
```

6.5.2. La méthode sqrt(double)

Cette méthode calcule la racine carrée de son paramètre.

Exemple (code java 1.1) :

```
public static void main(java.lang.String[] args) {
  System.out.println(" racine carree de 25  = "+Math.sqrt(25.0) );
}
```

Résultat :

```
racine carree de 25  = 5.0
```

6.5.3. La méthode exp(double)

Cette méthode calcule l'exponentielle de l'argument

Exemple (code java 1.1) :

```
public static void main(java.lang.String[] args) {
        System.out.println(" exponentiel de 5  = "+Math.exp(5.0) );
}
```

Résultat :

```
exponentiel de 5  = 148.4131591025766
```

6.5.4. La méthode log(double)

Cette méthode calcule le logarithme naturel de l'argument

Exemple (code java 1.1) :

```
public static void main(java.lang.String[] args) {
        System.out.println(" logarithme de 5  = "+Math.log(5.0) );
}
```

Résultat :

```
logarithme de 5  = 1.6094379124341003
```

6.6. La génération de nombres aléatoires

6.6.1. La méthode random()

Cette méthode renvoie un nombre aléatoire compris entre 0.0 et 1.0.

Exemple (code java 1.1) :

```
public static void main(java.lang.String[] args) {
        System.out.println(" un nombre aléatoire  = "+Math.random() );
}
```

Résultat :

```
un nombre aléatoire  = 0.8178819778125899
```

7. La gestion des exceptions

Les exceptions représentent le mécanisme de gestion des erreurs intégré au langage java. Il se compose d'objets représentant les erreurs et d'un ensemble de trois mots clés qui permettent de détecter et de traiter ces erreurs (try, catch et finally) et de les lever ou les propager (throw et throws).

Lors de la détection d'une erreur, un objet qui hérite de la classe Exception est créé (on dit qu'une exception est levée) et propagé à travers la pile d'execution jusqu'à ce qu'il soit traitée.

Ces mécanismes permettent de renforcer la sécurité du code java.

Exemple (code java 1.1) : une exception levée à l'exécution non capturée

```
public class TestException {
   public static void main(java.lang.String[] args) {
      int i = 3;
      int j = 0;
      System.out.println("résultat = " + (i / j));
   }
}
```

Résultat :

```
C:>java TestException
Exception in thread "main" java.lang.ArithmeticException: /
by zero
        at tests.TestException.main(TestException.java:23)
```

Si dans un bloc de code on fait appel à une méthode qui peut potentiellement générer une exception, on doit soit essayer de la récupérer avec try/catch, soit ajouter le mot clé throws dans la déclaration du bloc. Si on ne le fait pas, il y a une erreur à la compilation. Les erreurs et exceptions du paquetage java.lang échappe à cette contrainte. Throws permet de déléguer la responsabilité des erreurs vers la méthode appelante

Ce procédé présente un inconvénient : de nombreuses méthodes des packages java indiquent dans leur déclaration qu'elles peuvent lever une exception. Cependant ceci garantie que certaines exceptions critiques seront prises explicitement en compte par le programmeur.

7.1. Les mots clés try, catch et finally

Le bloc try rassemble les appels de méthodes susceptibles de produire des erreurs ou des exceptions. L'instruction try est suivie d'instructions entre des accolades.

Exemple (code java 1.1) :

```
    try {
        operation_risquée1;
        opération_risquée2;
    } catch (ExceptionInteressante e) {
        traitements
    } catch (ExceptionParticulière e) {
        traitements
    } catch (Exception e) {
        traitements
    } finally {
        traitement_pour_terminer_proprement;
    }
```

Si un événement indésirable survient dans le bloc try, la partie éventuellement non exécutée de ce bloc est abandonnée et le premier bloc catch est traité. Si catch est défini pour capturer l'exception issue du bloc try alors elle est traitée en exécutant le code associé au bloc. Si le bloc catch est vide (aucunes instructions entre les accolades) alors l'exception capturée est ignorée.

Si il y a plusieurs type d'erreurs et d'exceptions à intercepter, il faut définir autant de bloc catch que de type d'événement . Par type d'exception, il faut comprendre « qui est du type de la classe de l'exception ou d'une de ces sous classes ». Ainsi dans l'ordre séquentiel des clauses catch, un type d'exception de ne doit pas venir après un type d'une exception d'une super classe. Il faut faire attention à l'ordre des clauses catch pour traiter en premier les exceptions les plus précises (sous classes) avant les exceptions plus générales. Un message d'erreur est émis par le compilateur dans le cas contraire.

Exemple (code java 1.1) : erreur à la compil car Exception est traité en premier alors que ArithmeticException est une sous classe de Exception

```
public class TestException {
    public static void main(java.lang.String[] args) {
        // Insert code to start the application here.
        int i = 3;
        int j = 0;
        try {
            System.out.println("résultat = " + (i / j));
        }
        catch (Exception e) {
        }
        catch (ArithmeticException e) {
        }
    }
}
```

Résultat :

```
C:\tests>javac TestException.java
TestException.java:11: catch not reached.
        catch (ArithmeticException e) {
        ^
1 error
```

Si l'exception générée est une instance de la classe déclarée dans la clause catch ou d'une classe dérivée, alors on exécute le bloc associé. Si l'exception n'est pas traitées par un bloc catch, elle sera transmise au bloc de niveau supérieur. Si l'on ne se trouve pas dans un autre bloc try, on quitte la méthode en cours, qui regénère à son tour une exception dans la méthode appelante.

L'exécution totale du bloc try et d'un bloc d'une clause catch sont mutuellement exclusives : si une exception est levée, l'exécution du bloc try est arrêtée et si elle existe, la clause catch adéquate est exécutée.

La clause finally définit un bloc qui sera toujours exécuté, qu'une exception soit levée ou non. Ce bloc est facultatif. Il est aussi exécuté si dans le bloc try il y a une instruction break ou continue.

7.2. La classe Throwable

Cette classe descend directement de Object : c'est la classe de base pour le traitements des erreurs.

Cette classe possède deux constructeurs :

Méthode	Rôle
Throwable()	
Throwable(String)	La chaine en paramètre permet de définir un message qui décrit l'exception et qui pourra être consultée dans un bloc catch.

Les principales méthodes de la classe Throwable sont :

Méthodes	Rôle	Deprecated
String getMessage( )	lecture du message	
void printStackTrace( )	affiche l'exception et l'état de la pile d'execution au moment de son appel	
void printStackTrace(PrintStream s)	Idem mais envoie le résultat dans un flux	

Exemple (code java 1.1) :

```
public class TestException {
    public static void main(java.lang.String[] args) {
        // Insert code to start the application here.
        int i = 3;
        int j = 0;
        try {
            System.out.println("résultat = " + (i / j));
        }
        catch (ArithmeticException e) {
            System.out.println("getmessage");
            System.out.println(e.getMessage());
            System.out.println(" ");
            System.out.println("toString");
            System.out.println(e.toString());
            System.out.println(" ");
            System.out.println("printStackTrace");
            e.printStackTrace();
        }
    }
}
```

Résultat :

```
C:>java TestException
getmessage
/ by zero

toString
java.lang.ArithmeticException: / by zero

printStackTrace
java.lang.ArithmeticException: / by zero
        at tests.TestException.main(TestException.java:24)
```

7.3. Les classes Exception, RunTimeException et Error

Ces trois classes descendent de Throwable : en fait, toutes les exceptions dérivent de la classe Throwable.

La classe Error représente une erreur grave intervenue dans la machine virtuelle Java ou dans un sous système Java. L'application Java s'arrête instantanement dès l'apparition d'une exception de la classe Error.

La classe Exception représente des erreurs moins graves. La classe RuntimeException n'ont pas besoin d'être détectées impérativement par des blocs try/catch

7.4. Les exceptions personnalisées

Pour générer une exception, il suffit d'utiliser le mot clé throw, suivi d'un objet dont la classe dérive de Thowable. Si l'on veut générer une exception dans une méthode avec throw, il faut l'indiquer dans la déclaration de la méthode, en utilisant le mot clé throws.

En cas de nécessité, on peut créer ces propres exceptions. Elles descendent des classes Exception ou RunTimeException mais pas de la classe Error. Il est préférable (par convention) d'inclure le mot « Exception » dans le nom de la nouvelle classe.

Exemple (code java 1.1) :

```
public class SaisieErroneeException extends Exception {
    public SaisieErroneeException() {
        super();
    }
    public SaisieErroneeException(String s) {
        super(s);
    }
}

public class TestSaisieErroneeException {
    public static void controle(String chaine) throws
SaisieErroneeException {
        if (chaine.equals("") == true)
            throw new SaisieErroneeException("Saisie erronee : chaine vide");
    }
    public static void main(java.lang.String[] args) {
        String chaine1 = "bonjour";
        String chaine2 = "";
        try {
            controle(chaine1);
        }
        catch (SaisieErroneeException e) {
            System.out.println("Chaine1 saisie erronee");
        };
        try {
            controle(chaine2);
        }
        catch (SaisieErroneeException e) {
            System.out.println("Chaine2 saisie erronee");
        };
    }
}
```

Les méthodes pouvant lever des exceptions doivent inclure une clause throws nom_exception dans leur en tête. L'objectif est double : avoir une valeur documentaire et préciser au compilateur que cette méthode pourra lever cette exception et que toute méthode qui l'appelle devra prendre en compte cette exception (traitement ou propagation).

Si la méthode appelante ne traite pas l'erreur ou ne la propage pas, le compilateur génère l'exception nom_exception must be caught or it must be déclared in the thows clause of this méthode.

Java n'oblige la déclaration des exceptions dans l'en tête de la méthode que pour les exceptions dites controlées (checked). Les exception non controlées (unchecked) peuvent être capturée mais n'ont pas a être déclarée. Les exceptions et erreurs qui héritent de RunTimeException et de Error sont non controlées. Toutes les autres exceptions sont controlées.

8. Le multitâche

Ce chapitre est en cours d'écriture

Partie 2 : les interfaces graphiques

Les interfaces graphiques assurent le dialogue entre les utilisateurs et une application.

Dans un premier temps, java propose l'API AWT pour créer des interfaces graphiques. Depuis, java propose une nouvelle API nommée Swing.

Ces deux API peuvent être utilisées pour développer des applications ou des applets.

Cette partie contient les chapitres suivants :

- le graphisme : entame une série de chapitres sur les l'interfaces graphiques en détaillant les objets et méthodes de base pour le graphisme
- les éléments de l'AWT : recense les différents composants qui sont fournis dans l'AWT
- la création d'interfaces graphiques avec AWT : indique comment réaliser des interfaces graphiques avec l'AWT
- l'interception des actions de l'utilisateur : détaille les mécanismes qui permettent de réagir aux actions de l'utilisateur via une interface graphique
- le développement d'interface graphique avec Swing : indique comment réaliser des interfaces graphiques avec Swing
- les applets : plonge au coeur des premières applications qui ont rendu java célèbre

9. Le graphisme

La classe Graphics contient les outils nécessaires pour dessiner. Cette classe est abstraite et elle ne possède pas de constructeur public : il n'est pas possible de construire des instances de graphics nous même. Les instances nécessaires sont fournies par le système d'exploitation qui instanciera via à la machine virtuelle une sous classe de Graphics dépendante de la plateforme utilisée.

9.1 Les opérations sur le contexte graphique

9.1.1 Le tracé de formes géométriques

A l 'exception des lignes, toutes les formes peuvent être dessinées vides (méthode drawXXX) ou pleines (fillXXX).

La classe Graphics possède de nombreuses méthodes qui permettent de réaliser des dessins.

Méthode	Role
drawRect(x, y, largeur, hauteur) fillRect(x, y, largeur, hauteur)	dessiner un carré ou un rectangle
drawRoundRect(x, y, largeur, hauteur, hor_arr,ver_arr) fillRoundRect(x, y, largeur, hauteur, hor_arr, ver_arr)	dessiner un carré ou un rectangle arrondi
drawLine(x1, y1, x2, y2)	Dessiner une ligne
drawOval(x, y, largeur, hauteur) fillOval(x, y, largeur, hauteur)	dessiner un cercle ou une elipse en spécifiant le rectangle dans lequel ils s'incrivent
drawPolygon(int[], int[], int) fillPolygon(int[], int[], int)	Dessiner un polygone ouvert ou fermé. Les deux premiers paramètres sont les coordonnées en abscisses et en ordonnées. Le dernier paramètres est le nombre de points du polygone. Pour dessiner un polygone fermé il faut joindre le dernier point au premier. Exemple (code java 1.1) : `int[] x={10,60,100,80,150,10};` `int[] y={15,15,25,35,45,15};` `g.drawPolygon(x,y,x.length);` `g.fillPolygon(x,y,x.length);` Il est possible de définir un objet Polygon.

	Exemple (code java 1.1) : `int[] x={10,60,100,80,150,10};` `int[] y={15,15,25,35,45,15};` `Polygon p = new Polygon(x, y,x.length );` `g.drawPolygon(p);`
drawArc(x, y, largeur, hauteur, angle_deb, angle_bal) fillArc(x, y, largeur, hauteur, angle_deb, angle_bal);	dessiner un arc d'ellipse inscrit dans un rectangle ou un carré. L'angle 0 se situe à 3 heures. Il faut indiquer l'angle de début et l'angle balayé

9.1.2 Le tracé de texte

La méthode drawString() permet d'afficher un texte aux coordonnées précisées

Exemple (code java 1.1) :

```
g.drawString(texte, x, y );
```

Pour afficher des nombres int ou float, il suffit de les concatener à une chaine éventuellement vide avec l'opérateur +.

9.1.3 L'utilisation des fontes

La classe Font permet d'utiliser une police de caractères particulière pour affiche un texte.

Exemple (code java 1.1) :

```
Font fonte = new Font(« TimesRoman »,Font.BOLD,30);
```

Le constructeur de la classe Font est Font(String, int, int). Les paramètres sont : le nom de la police, le style (BOLD, ITALIC, PLAIN ou 0,1,2) et la taille des caractères en points.

Pour associer plusieurs styles, il suffit de les additionner

Exemple (code java 1.1) :

```
Font.BOLD + Font.ITALIC
```

Si la police spécifiée n'existe pas, Java prend la fonte par défaut même si une autre a été spécifiée précedemment. Le style et la taille seront tout de même adpatés. La méthode getName() de la classe Font retourne le nom de la fonte.

La méthode setFont() de la classe Graphics permet de changer la police d'affichage des textes

Exemple (code java 1.1) :

```
Font fonte = new Font("
TimesRoman ",Font.BOLD,30);
g.setFont(fonte);
g.drawString("bonjour",50,50);
```

Les polices suivantes sont utilisables : Dialog, Helvetica, TimesRoman, Courier, ZapfDingBats

9.1.4 La gestion de la couleur

La méthode setColor() permet de fixer la couleur des éléments graphiques des objets de type Graphics créés après à son appel.

Exemple (code java 1.1) :
`g.setColor(Color.black); //(green, blue, red, white, black, ...)`

9.1.5 Le chevauchement de figures graphiques

Si 2 surfaces de couleur différentes se superposent, alors la derniere dessinée recouvre la précédente sauf si on invoque la méthode setXORMode(). Dans ce cas, la couleur de l'intersection prend une autre couleur. L'argument à fournir est une couleur alternative. La couleur d'intersection représente une combinaison de la couleur originale et de la couleur alternative.

9.1.6 L'effacement d'une aire

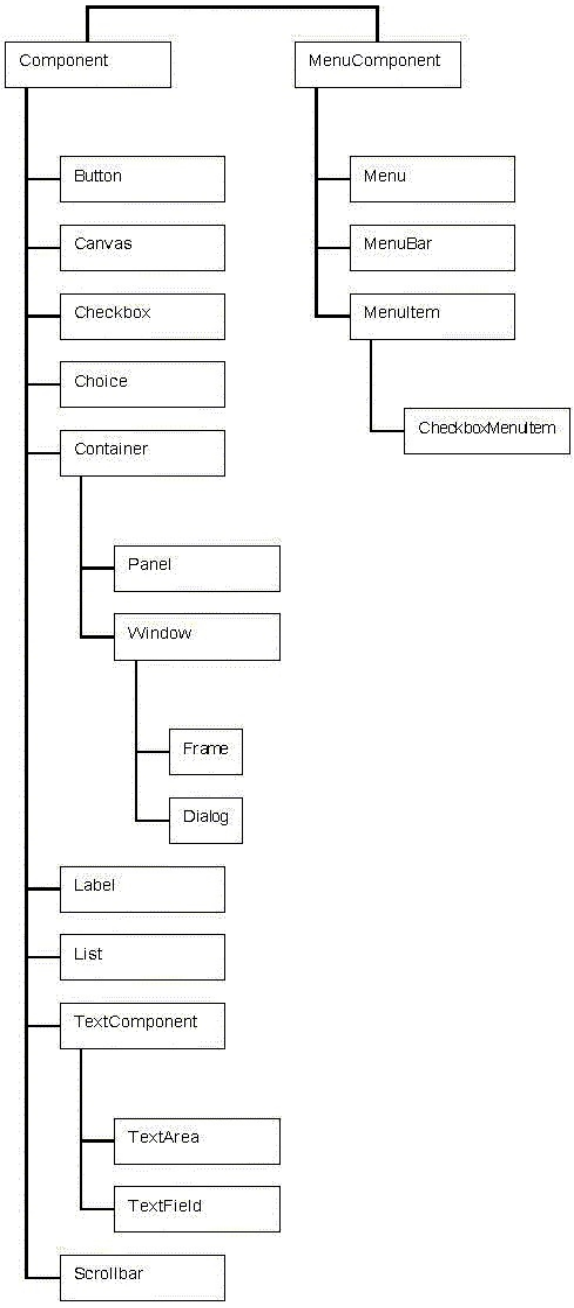
La méthode clearRect(x1, y1, x2, y2) dessine un rectangle dans la couleur de fond courante.

9.1.7 La copier une aire rectangulaire

La méthode copyArea(x1, y1, x2, y2, dx, dy) permet de copier une aire rectangulaire. Les paramètres dx et dy permettent de spécifier un décalage en pixels de la copie par rapport à l'originale.

10. Les éléments d'interface graphique de l'AWT

Les classes du toolkit AWT (Abstract Windows Toolkit) permettent d'écrire des interfaces graphiques indépendantes du système d'exploitation sur lesquelles elles vont fonctionner. Cette librairie utilise le système graphique de la plateforme d'exécution (Windows, MacOS, X−Window) pour afficher les objets graphiques. Le toolkit contient des classes décrivant les composants graphiques, les polices, les couleurs et les images.



Le diagramme ci dessus définit une vue partielle de la hiérarchie des classes (les relations d'héritage) qu'il ne faut pas confondre avec la hiérarchie interne à chaque application qui définit l'imbrication des différents composants graphiques.

Les deux classes principales de AWT sont Component et Container. Chaque type d'objet de l'interface graphique est une classe dérivée de Component. La classe Container, qui hérite de Component est capable de contenir d'autres objets graphiques (tout objet dérivant de Component).

Ce chapitre contient plusieurs sections :

10.1. Les composants graphiques

Pour utiliser un composant, il faut créer un nouvel objet représentant le composant et l'ajouter à un de type conteneur qui existe avec la méthode add().

Exemple (code java 1.1) : ajout d'un bouton dans une applet (Applet hérite de Panel)

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

public class AppletButton extends Applet {

   Button b = new Button(" Bouton ");

   public void init() {
      super.init();
      add(b);
   }
}
```

10.1.1. Les étiquettes

Il faut utiliser un objet de la classe java.awt.Label

Exemple (code java 1.1) :

```
Label la = new Label( );
la.setText("une etiquette");
// ou Label la = new Label("une etiquette");
```

Il est possible de créer un objet de la classe java.awt.Label en précisant l'alignement du texte

Exemple (code java 1.1) :

```
Label la = new Label("etiquette", Label.RIGHT);
```

Le texte à afficher et l'alignement peuvent être modifiés dynamiquement lors de l'éxecution :

Exemple (code java 1.1) :

```
la.setText("nouveau texte");
la.setAlignment(Label.LEFT);
```

10.1.2. Les boutons

Il faut utiliser un objet de la classe java.awt.Button

Cette classe possède deux constructeurs :

Constructeur	Rôle
Button()	
Button(String)	Permet de préciser le libellé du bouton

Exemple (code java 1.1) :

```
Button bouton = new Button();
bouton.setLabel("bouton");
// ou Button bouton = new Button("bouton");
```

Le libellé du bouton peut être modifié dynamiquement grace à la méthode setLabel() :

Exemple (code java 1.1) :

```
bouton.setLabel("nouveau libellé");
```

10.1.3. Les panneaux

Les panneaux sont des conteneurs qui permettent de rassembler des composants et de les positionner grace à un gestionnaire de présentation. Il faut utiliser un objet de la classe java.awt.Panel.

Par défaut le gestionnaire de présentation d'un panel est de type FlowLayout.

Constructeur	Role

Panel()	Créer un panneau avec un gestionnaire de présentation de type FlowLayout
Panel(LayoutManager)	Créer un panneau avec le gestionnaire précisé en paramètre

Exemple (code java 1.1) :

```
Panel p = new Panel();
```

L'ajout d'un composant au panel se fait grace à la méthode add().

Exemple (code java 1.1) :

```
p.add(new Button("bouton");
```

10.1.4. Les listes déroulantes (combobox)

Il faut utiliser un objet de la classe java.awt.Choice

Cette classe ne possède qu'un seul constructeur qui ne possèdent pas de paramètres.

Exemple (code java 1.1) :

```
Choice maCombo = new Choice();
```

Les méthodes add() et addItem() permettent d'ajouter des éléments à la combo.

Exemple (code java 1.1) :

```
maCombo.addItem("element 1");
// ou maCombo.add("element 2");
```

Plusieurs méthodes permettent la gestion des sélections :

Méthodes	Role	Deprecated
void select(int);	sélectionner un élément par son indice : le premier élément correspond à l'indice 0. Une exception IllegalArgumentException est levée si l'indice ne correspond pas à un élément. Exemple (code java 1.1) : `maCombo.select(0);`	
void select(String);	sélectionner un élément par son contenu	

	Aucune exception est levée si la chaine de caractères ne correspond à aucun élément : l'élément sélectionné ne change pas. Exemple (code java 1.1) : `maCombo.select("element 1");`	
int countItems( );	déterminer le nombre d'élément de la liste. La méthode countItems() permet d'obtenir le nombre d'éléments de la combo. Exemple (code java 1.1) : `int n;` `n=maCombo.countItems();`	 il faut utiliser getItemCount() à la place Exemple (code java 1.1) : `int n;` `n=maCombo.getItemCount();`
String getItem(int);	lire le contenu de l'élément d'indice n Exemple (code java 1.1) : `String c = new String( );` `c = maCombo.getItem(n);`	
String getSelectedItem();	déterminer le contenu de l'élément sélectionné Exemple (code java 1.1) : `String s = new String( );` `s = maCombo.getSelectedItem();`	
int getSelectedIndex( );	déterminer l'index de l'élément selectionné Exemple (code java 1.1) : `int n;` `n=maCombo.getSelectedIndex();`	

10.1.5. La classe TextComponent

La classe TextComponent est la classe des mères des classes qui permettent l'édition de texte : TextArea et TextField.

Elle définit un certain nombre de méthodes dont ces classes héritent.

Méthodes	Role	Deprecated
String getSelectedText( );	Renvoie le texte sélectionné	
int getSelectionStart( );	Renvoie la position de début de sélection	

int getSelectionEnd( );	Renvoie la position de fin de sélection	
String getText( );	Renvoie le texte contenu dans l'objet	
boolean isEditable( );	Retourne un booleen indiquant si le texte est modifiable	
void select(int start, int end);	Sélection des caractères situés entre start et end	
void selectAll( );	Sélection de tout le texte	
void setEditable(boolean b);	Autoriser ou interdire la modification du texte	
void setText(String s);	Définir un nouveau texte	

10.1.6. Les champs de texte

Il faut déclarer un objet de la classe java.awt.TextField

Il existe plusieurs constructeurs :

Constructeurs	Role
TextField();	
TextField(int);	prédetermination du nombre de caractères à saisir
TextField(String);	avec texte par défaut
TextField(String, int);	avec texte par défaut et nombre de caractères à saisir

Cette classe possède quelques méthodes utiles :

Méthodes	Role	Deprecated
String getText( )	lecture de la chaine saisie Exemple (code java 1.1) : `String saisie = new String();` `saisie = tf.getText( );`	
int getColumns( )	lecture du nombre de caractères predéfini Exemple (code java 1.1) : `int i;` `i = tf.getColumns( );`	
void setEchoCharacter()	pour la saisie d'un mot de passe : remplace chaque caractère saisi par celui fourni en paramètre Exemple (code java 1.1) :	 il faut utiliser la méthode setEchoChar() Exemple (code java 1.1) :

	`tf.setEchoCharacter('*');` `TextField tf = new TextField(10);`	`tf.setEchoChar('*');`

10.1.7. Les zones de texte multilignes

Il faut déclarer un objet de la classe java.awt.TextArea

Il existe plusieurs constructeurs :

Constructeur	Role
TextArea()	
TextArea(int, int)	avec prédetermination du nombre de lignes et de colonnes
TextArea(String)	avec texte par défaut
TextArea(String, int, int)	avec texte par défaut et taille

Les principales méthodes sont :

Méthodes	Role	Deprecated
String getText()	lecture du contenu intégral de la zone de texte Exemple (code java 1.1) : `String contenu = new String;` `contenu = ta.getText( );`	
String getSelectedText( )	lecture de la portion de texte sélectionnée Exemple (code java 1.1) : `String contenu = new String;` `contenu = ta.getSelectedText( );`	
int getRows()	détermination du nombre de lignes Exemple (code java 1.1) : `int n;` `n = ta.getRows( );`	
int getColumns( )	détermination du nombre de colonnes Exemple (code java 1.1) : `int n;` `n = ta.getColumns( );`	

<table>
<tr>
<td>void insertText(String, int)</td>
<td>insertion de la chaine à la position fournie
Exemple (code java 1.1) :
<pre>String text =
new String(«texte inséré»);
int n =10;
ta.insertText(text,n);</pre></td>
<td> 
Il faut utiliser la méthode insert()
Exemple (code java 1.1) :
<pre>String text = new
String(«texte inséré»);
int n =10;
ta.insert(text,n);</pre></td>
</tr>
<tr>
<td>void setEditable(boolean)</td>
<td>Autoriser la modification
Exemple (code java 1.1) :
<pre>ta.setEditable(False);
//texte non modifiable</pre></td>
<td></td>
</tr>
<tr>
<td>void appendText(String)</td>
<td>Ajouter le texte transmis au texte existant
Exemple (code java 1.1) :
<pre>ta.appendTexte(String text);</pre></td>
<td> 
Il faut utiliser la méthode append()</td>
</tr>
<tr>
<td>void replaceText(String, int, int)</td>
<td>Remplacer par text le texte entre les positions start et end
Exemple (code java 1.1) :
<pre>ta.replaceText(text, 10, 20);</pre></td>
<td> 
il faut utiliser la méthode replaceRange()</td>
</tr>
</table>

10.1.8. Les listes

Il faut déclarer un objet de la classe java.awt.List.

Il existe plusieurs constructeurs :

Constructeur	Role
List( )	
List(int)	Permet de préciser le nombre de lignes affichées
List(int, boolean)	Permet de préciser le nombre de lignes affichées et l'indicateur de sélection multiple

Les principales méthodes sont :

Méthodes	Role	Deprecated

void addItem(String)	ajouter un élément Exemple (code java 1.1) : `li.addItem("nouvel element");` `// a jout en fin de liste`	 il faut utiliser la méthode add()
void addItem(String, int)	insérer un élément à un certain emplacement : le premier element est en position 0 Exemple (code java 1.1) : `li.addItem("ajout ligne",2);`	 il faut utiliser la méthode add()
void delItem(int)	retirer un élément de la liste Exemple (code java 1.1) : `li.delItem(0);` `// supprime le premier element`	 il faut utiliser la méthode remove()
void delItems(int, int)	supprimer plusieurs éléments consécutifs entre les deux indices Exemple (code java 1.1) : `li.delItems(1, 3);`	 cette méthode est deprecated
void clear()	effacement complet du contenu de la liste Exemple (code java 1.1) : `li.clear( );`	 il faut utiliser la méthode removeAll()
void replaceItem(String, int)	remplacer un élément Exemple (code java 1.1) : `li.replaceItem(` `"ligne remplacee", 1);`	
int countItems()	nombre d'élément de la liste Exemple (code java 1.1) : `int n;` `n = li.countItems( );`	 il faut utiliser la méthode getItemCount()
int getRows()	nombre de ligne de la liste Exemple (code java 1.1) :	

	Exemple (code java 1.1) : `int n;` `n = li.getRows( );`	
String getItem(int)	contenu d'un élément Exemple (code java 1.1) : `String text = new String( );` `text = li.getItem(1);`	
void select(int)	sélectionner un élément Exemple (code java 1.1) : `li.select(0);`	
setMultipleSelections(boolean)	déterminer si la sélection multiple est autorisée Exemple (code java 1.1) : `li.setMultipleSelections(true);`	 il faut utiliser la méthode setMultipleMode()
void deselect(int)	désélectionner un élément Exemple (code java 1.1) : `li.deselect(0);`	
int getSelectedIndex( )	déterminer l'élément sélectionné en cas de selection simple : renvoi l'indice ou −1 si aucun element n est selectionne Exemple (code java 1.1) : `int i;` `i = li.getSelectedIndex();`	
int[] getSelectedIndexes( )	déterminer les éléments sélectionnées en cas de sélection multiple Exemple (code java 1.1) : `int i[]=li.getSelectedIndexes();`	
String getSelectedItem( )	déterminer le contenu en cas de sélection simple : renvoi le texte ou null si pas de selection Exemple (code java 1.1) :	

	`` String texte = new String( ); texte = li.getSelectedItem( ); ```	
String[] getSelectedItems()	déterminer les contenus des éléments sélectionnés en cas de sélection multiple : renvoi les textes sélectionnées ou null si pas de sélection Exemple (code java 1.1) : ``` String texte[ ] = li.getSelectedItems(); for (i = 0 ; i < texte.length(); i++) System.out.println(texte[i]); ```	
boolean isSelected(int)	déterminer si un élément est sélectionné Exemple (code java 1.1) : ``` boolean selection; selection = li.isSelected(0); ```	 il faut utiliser la méthode isIndexSelect()
int getVisibleIndex()	renvoie l'index de l'entrée en haut de la liste Exemple (code java 1.1) : ``` int top = li.getVisibleIndex(); ```	
void makeVisible(int)	assure que l'élement précisé sera visible Exemple (code java 1.1) : ``` li.makeVisible(10); ```	

Exemple (code java 1.1) : une liste de 5 éléments avec sélection multiple

```
import java.awt.*;

class TestList {

  static public void main (String arg [ ]) {

    Frame frame = new Frame("Une liste");

    List list = new List(5,true);
    list.add("element 0");
    list.add("element 1");
    list.add("element 2");
    list.add("element 3");
    list.add("element 4");

    frame.add(List);
    frame.show();
    frame.pack();
  }
}
```

10.1.9. Les cases à cocher

Il faut déclarer un objet de la classe java.awt.Checkbox

Il existe plusieurs constructeurs :

Constructeur	Role
Checkbox( )	
Checkbox(String)	avec une étiquette
Checkbox(String,boolean)	avec une étiquette et un état
Checkbox(String,CheckboxGroup, boolean)	avec une étiquette, dans un groupe de cases à cocher et un état

Les principales méthodes sont :

Méthodes	Role	Deprecated
void setLabel(String)	modifier l'étiquette Exemple (code java 1.1) : `cb.setLabel( "libelle de la case : " );`	
void setState(boolean)	fixer l'état Exemple (code java 1.1) : `cb.setState( true );`	
boolean getState( )	consulter l'état de la case Exemple (code java 1.1) : `boolean etat;` `etat = cb.getState( );`	
String getLabel( )	lire l'étiquette de la case Exemple (code java 1.1) : `String commentaire = new String( );` `commentaire = cb.getLabel( );`	

10.1.10. Les boutons radio

Déclarer un objet de la classe java.awt.CheckboxGroup

Exemple (code java 1.1) :

```
CheckboxGroup rb;
Checkbox cb1 = new Checkbox(« etiquette 1 », rb, etat1_boolean);
Checkbox cb2 = new Checkbox(« etiquette 2 », rb, etat1_boolean);
Checkbox cb3 = new Checkbox(« etiquette 3 », rb, etat1_boolean);
```

Les principales méthodes sont :

Méthodes	Role	Deprecated
Checkbox getCurrent()	retourne l'objet Checkbox correspondant à la réponse sélectionnée	 il faut utiliser la méthode getSelectedCheckbox()
void setCurrent(Checkbox)	Coche le bouton radio passé en paramètre	 il faut utiliser la méthode setSelectedCheckbox()

10.1.11. Les barres de défilement

Il faut déclarer un objet de la classe java.awt.Scrollbar

Il existe plusieurs constructeurs :

Constructeur	Role
Scrollbar( )	
Scrollbar(orientation)	
Scrollbar(orientation, valeur_initiale, visible, min, max)	

orientation : Scrollbar.VERTICALE ou Scrollbar.HORIZONTAL

valeur_initiale : position du curseur à la création

visible : taille de la partie visible de la zone défilante

min : valeur minimale associée à la barre

max : valeur maximale associée à la barre

Les principales méthodes sont :

Méthodes	Role	Deprecated
sb.setValues(int,int,int,int)	maj des parametres de la barre Exemple (code java 1.1) :	

	sb.setValues(valeur, visible, minimum, maximum);	
void setValue(int)	modifer la valeur courante Exemple (code java 1.1) : sb.setValue(10);	
int getMaximum( );	lecture du maximum Exemple (code java 1.1) : int max = sb.getMaximum( );	
int getMinimum( );	lecture du minimum Exemple (code java 1.1) : int min = sb.getMinimum( );	
int getOrientation( )	lecture de l'orientation Exemple (code java 1.1) : int o = sb.getOrientation( );	
int getValue( );	lecture de la valeure courante Exemple (code java 1.1) : int valeur = sb.getValue( );	
void setLineIncrement(int);	détermine la valeur à ajouter ou à oter quand l'utilisateur clique sur une flèche de défilement	 il faut utiliser la méthode setUnitIncrement()
int setPageIncrement( );	détermine la valeur à ajouter ou à oter quand l'utilisateur clique sur le conteneur	 il faut utiliser la méthode setBlockIncrement()

10.1.12. La classe Canvas

C'est un composant sans fonction particulière : il est utile pour créer des composants graphiques personnalisés.

Il est nécessaire d'étendre la classe Canvas pour en redéfinir la méthode Paint().

syntaxe : Cancas can = new Canvas( );

Exemple (code java 1.1) :

```
import java.awt.*;

public class MonCanvas extends Canvas {

   public void paint(Graphics g) {
      g.setColor(Color.black);
      g.fillRect(10, 10, 100,50);
      g.setColor(Color.green);
      g.fillOval(40, 40, 10,10);
   }

}


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

public class AppletButton extends Applet {

   MonCanvas mc = new MonCanvas();

   public void paint(Graphics g) {
      super.paint(g);
      mc.paint(g);
   }
}
```

10.2. La classe Component

Les contrôles fenêtrés descendent plus ou moins directement de la classe AWT Component.

Cette classe contient de nombreuse méthodes :

Méthodes	Role	Deprecated
Rectangle bounds()	renvoie la position actuelle et la taille des composants	 utiliser la méthode getBounds().
void disable()	désactive les composants	 utiliser la méthode setEnabled(false).
void enable()	active les composants	 utiliser la méthode setEnabled(true).
void enable(boolean)	active ou désactive le composant selon la valeur du paramètre	 utiliser la méthode setEnabled(boolean).
Color getBackGround()	renvoie la couleur actuelle d'arrière plan	

Font getFont()	renvoie la fonte utilisée pour afficher les caractères	
Color getForeGround()	renvoie la couleur de premier plan	
Graphics getGraphics()	renvoie le contexte graphique	
Container getParent()	renvoie le conteneur (composant de niveau supérieure)	
void hide()	masque l'objet	 utiliser la méthode setVisible().
boolean inside(int x, int y)	indique si la coordonnée écran absolue se trouve dans l'objet	 utiliser la méthode contains().
boolean isEnabled()	indique si l'objet est actif	
boolean isShowing()	indique si l'objet est visible	
boolean isVisible()	indique si l'objet est visible lorque sont conteneur est visible	
boolean isShowing()	indique si une partie de l'objet est visible	
void layout()	repositionne l'objet en fonction du Layout Manager courant	 utiliser la méthode doLayout().
Component locate(int x, int y)	retourne le composant situé à cet endroit	 utiliser la méthode getComponentAt().
Point location()	retourne l'origine du composant	 utiliser la méthode getLocation().
void move(int x, int y)	déplace les composants vers la position spécifiée	 utiliser la méthode setLocation().
void paint(Graphics);	dessine le composant	
void paintAll(Graphics)	dessine le composant et ceux qui sont contenus en lui	
void repaint()	redessine le composant pat appel à la méthode update()	
void requestFocus();	demande le focus	

void reshape(int x, inty, int w, int h)	modifie la position et la taille (unité : points écran)	 utiliser la méthode setBounds().
void resize(int w, int h)	modifie la taille (unité : points écran)	 utiliser la méthode setSize().
void setBackground(Color)	définie la couleur d'arrière plan	
void setFont(Font)	définie la police	
void setForeground(Color)	définie la couleur de premier plan	
void show()	affiche le composant	 utiliser la méthode setVisible(True).
Dimension size()	détermine la taille actuelle	 utiliser la méthode getSize().

10.3. Les conteneurs

Les conteneurs sont des objets graphiques qui peuvent contenir d'autres objets graphiques, incluant éventuellement des conteneurs. Ils héritent de la classe Container.

Un composant graphique doit toujours être incorporer dans un conteneur :

Conteneur	Role
Panel	conteneur sans fenetre propre. Utile pour ordonner les controles
Window	fenêtre principale sans cadre ni menu. Les objets descendants de cette classe peuvent servir à implémenter des menus
Dialog (descendant de Window)	réaliser des boîtes de dialogue simples
Frame (descendant de Window)	classe de fenêtre completement fonctionnelle
Applet (descendant de Panel)	pas de menu. Pas de boîte de dialogue sans être incorporée dans une classe Frame.

L'insertion de composant dans un conteneur se fait grace à la méthode add(Component) de la classe Container.

Exemple (code java 1.1) :

```
Panel p = new Panel();

Button b1 = new button(« premier »);
p.add(b1);
Button b2;
p.add(b2 = new Button (« Deuxième »);
```

```
p.add(new Button(«Troisième »);
```

10.3.1. Le conteneur Panel

C'est essentiellement un objet de rangement pour d'autres composants.

La classe Panel possède deux constructeurs :

Constructeur	Role
Panel()	
Panel(LayoutManager)	Permet de préciser un layout manager

Exemple (code java 1.1) :

```
Panel p = new Panel( );

Button b = new Button(« bouton »);
p.add( b);
```

10.3.2. Le conteneur Window

La classe Window contient plusieurs méthodes dont voici les plus utiles :

Méthodes	Role	Deprecated
void pack()	Calculer la taille et la position de tous les controles de la fenetre. La méthode pack() agit en étroite collaboration avec le layout manager et permet à chaque contrôle de garder, dans un premier temps sa taille optimale. Une fois que tous les contrôles ont leur taille optimale, pack() utilise ces informations pour positionner les contrôles. pack() calcule ensuite la taille de la fenêtre. L'appel à pack() doit se faire à l'intérieur du constructeur de fenêtre après insertion de tous les contrôles.	
void show()	Afficher la fenetre	
void dispose()	Liberer les ressources allouée à la fenetre	

10.3.3. Le conteneur Frame

Permet de créer des fenêtre d'encadrement. Il hérite de Window qui ne s'occupe que de l'ouverture de la fenêtre. Window ne connait pas les menus ni les bordures qui sont gérés par Frame. Dans une applet, elle n'apparait pas dans le navigateur mais comme un fenêtre indépendante.

Il existe deux constructeurs :

Constructeur	Role
Frame()	Exemple : Frame f = new Frame( );
Frame(String)	Precise le nom de la fenetre Exemple : Frame f = new Frame(« titre »);

Les principales méthodes sont :

Méthodes	Role	Deprecated
setCursor(int)	changer le pointeur de la souris dans la fenetre Exemple : f.setCursor(Frame.CROSSHAIR_CURSOR);	 utiliser la méthode setCursor(Cursor).
int getCursorType()	déterminer la forme actuelle du curseur	 utiliser la méthode getCursor().
Image getIconImage()	déterminer l'icone actuelle de la fenetre	
MenuBar getMenuBar()	déterminer la barre de menus actuelle	
String getTitle()	déterminer le titre de la fenêtre	
boolean isResizeable()	déteriner si la taille est modifiable	
void remove(MenuComponent)	Supprimer un menu	
void setIconImage(Image);	définir l'icone de la fenêtre	
void setMenuBar(MenuBar)	Définir la barre de menu	
void setResizeable(boolean)	définir si la taille peut être modifiée	
void SetTitle(String)	définir le titre de la fenêtre	

Exemple (code java 1.1) :

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

public class AppletFrame extends Applet {

   Frame f;

   public void init() {
      super.init();
      // insert code to initialize the applet here
      f = new Frame("titre");
      f.add(new Label("hello "));
      f.show();
      f.setSize(300, 100);
   }
}
```

Le message « Warning : Applet window » est impossible à enlever dans la fenetre : cela permet d'éviter la création d'une applet qui demande un mot de passe.

Le gestionnaire de mise en page par défaut d'une Frame est BorderLayout (FlowLayout pour une applet).

Exemple (code java 1.1) : Exemple : construction d'une fenêtre simple

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      show(); // affiche la fenetre
   }

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

10.3.4. Le conteneur Dialog

La classe Dialog hérite de la classe Window.

Une boîte de dialogue doit dérivée de la Classe Dialog de package java.awt.

Un objet de la classe Dialog doit dépendre d'un objet de la classe Frame.

Exemple (code java 1.1) :

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

public class Apropos extends Dialog {

   public APropos(Frame parent) {
      super(parent, "A propos ", true);
      addWindowListener(new
      AProposListener(this));
      setSize(300, 300);
      setResizable(False);
   }
}

class AProposListener extends WindowAdapter {

   Dialog dialogue;
   public AProposListener(Dialog dialogue) {
      this.dialogue = dialog;
   }

   public void windowClosing(WindowEvent e) {
      dialogue.dispose();
   }

}
```

L'appel du constructeur Dialog(Frame, String, Boolean) permet de créer une instance avec comme paramètres : la fenetre à laquelle appartient la boîte de dialogue, le titre de la boîte, le caractère modale de la boîte.

La méthode dispose() de la classe Dialog ferme la boîte et libère les ressources associées. Il ne faut pas associer cette action à la méthode windowClosed() car dispose provoque l'appel de windowClosed ce qui entrainerait un appel récursif infinie.

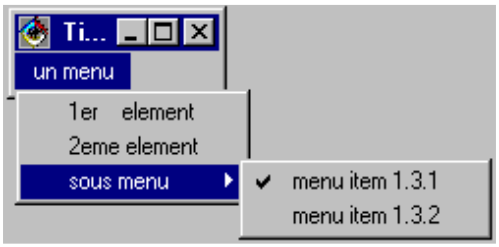
10.4. Les menus

Il faut insérer les menus dans des objets de la classe Frame (fenêtre d'encadrement). Il n'est donc pas possible d'insérer directement des menus dans une applet.

Il faut créer une barre de menu et l'affecter à la fenêtre d'encadrement. Il faut créer ensuite créer les entrées de chaque menu et les rattacher à la barre. Ajouter ensuite les éléments à chacun des menus.

Exemple (code java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);

      MenuBar mb = new MenuBar();
      setMenuBar(mb);

      Menu m = new Menu(" un menu ");
      mb.add(m);
      m.add(new MenuItem(" 1er element "));
      m.add(new MenuItem(" 2eme element "));
      Menu m2 = new Menu(" sous menu ");

      CheckboxMenuItem cbm1 = new CheckboxMenuItem(" menu item 1.3.1 ");
      m2.add(cbm1);
      cbm1.setState(true);
      CheckboxMenuItem cbm2 = new CheckboxMenuItem(" menu item 1.3.2 ");
      m2.add(cbm2);

      m.add(m2);

      pack();
      show(); // affiche la fenetre

   }

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

}
```



Exemple (code java 1.1) : création d'une classe qui définit un menu

```
import java.awt.*;

public class MenuFenetre extends java.awt.MenuBar {

   public MenuItem menuQuitter, menuNouveau, menuApropos;

   public MenuFenetre() {

      Menu menuFichier = new Menu(" Fichier ");
```

```
      menuNouveau = new MenuItem(" Nouveau ");
      menuQuitter = new MenuItem(" Quitter ");

      menuFichier.add(menuNouveau);

      menuFichier.addSeparator();

      menuFichier.add(menuQuitter);

      Menu menuAide = new Menu(" Aide ");
      menuApropos = new MenuItem(" A propos ");
      menuAide.add(menuApropos);

      add(menuFichier);

      setHelpMenu(menuAide);
   }
}
```

La méthode setHelpMenu() confère sous certaines plateformes un comportement particulier à ce menu.

La méthode setMenuBar() de la classe Frame prend en paramètre une instance de la classe MenuBar. Cette instance peut être directement une instance de la classe MenuBar qui aura été modifiée grace aux méthodes add() ou alors une classe dérivée de MenuBar qui est adaptée aux besoins (voir Exemple);

Exemple (code java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      MenuFenetre mf = new
      MenuFenetre();

      setMenuBar(mf);

      pack();

      show(); // affiche la fenetre

   }

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

}
```

10.4.1 Les méthodes de la classe MenuBar

Méthodes	Role	Deprecated

void add(Menu)	ajouter un menu dans la barre	
int countMenus()	renvoie le nombre de menus	 utiliser la méthode getMenuCount().
Menu getMenu(int pos)	renvoie le menu à la position spécifiée	
void remove(int pos)	supprimer le menu à la position spécifiée	
void remove(Menu)	supprimer le menu de la barre de menu	

10.4.2. Les méthodes de la classe Menu

Méthodes	Role	Deprecated
MenuItem add(MenuItem) void add(String)	ajouter une option dans le menu	
void addSeparator()	ajouter un trait de séparation dans le menu	
int countItems()	renvoie le nombre d'options du menu	 utiliser la méthode getItemCount().
MenuItem getItem(int pos)	déterminer l'option du menu à la position spécifiée	
void remove(MenuItem mi)	supprimer la commande spécifiée	
void remove(int pos)	supprimer la commande à la position spécifiée	

10.4.3. Les méthodes de la classe MenuItem

Méthodes	Role	Deprecated
void disable()	désactiver l'élément	 utiliser la méthode setEnabled(false).
void enable()	activer l'élément	 utiliser la méthode setEnabled(true).
void enable(boolean cond)	désactiver ou activer l'élément en fonction du paramètre	 utiliser la méthode setEnabled(boolean).
String getLabel()	Renvoie le texte de l'élément	
boolean isEnabled()	renvoie l'état de l'élément (actif / inactif)	

void setLabel(String text)	définir une nouveau texte pour la commande	

10.4.4. Les méthodes de la classe CheckboxMenuItem

Méthodes	Role	Deprecated
boolean getState()	renvoie l'état d'activation de l'élément	
Void setState(boolean)	définir l'état d'activation de l'élément	

11. La création d'interface graphique avec AWT

Ce chapitre contient plusieurs sections :

11.1. Le dimensionnement des composants

En principe, il est automatique grace au LayoutManager. Pour donner à un composant une taille donnée, il faut redéfinir la méthode getPreferedSize de la classe Component.

Exemple (code java 1.1) :

```
import java.awt.*;

public class MonBouton extends Button {

   public Dimension getPreferredSize() {
      return new Dimension(800, 250);
   }

}
```

Le méthode getPreferedSize() indique la taille souhaitée mais pas celle imposée. En fonction du Layout Manager, le composant pourra ou non imposer sa taille.

Layout	Hauteur	Largeur
Sans Layout	oui	oui
FlowLayout	oui	oui
BorderLayout(East, West)	non	oui
BorderLayout(North, South)	oui	non

BorderLayout(Center)	non	non
GridLayout	non	non

Cette méthode oblige à sous classer tous les composants.

Une autre façon de faire est de se passer des Layout et de placer les composants à la main en indiquant leurs coordonnées et leurs dimensions.

Pour supprimer le Layout par défaut d'une classe, il faut appeler la méthode setLayout() avec comme paramètre null.

Trois méthodes de la classe Component permettent de positionner des composants :

- setBounds(int x, int y, int largeur, int hauteur)
- setLocation(int x , int y)
- setSize(int largeur, int hauteur)

Ces méthodes permettent de placer un composant à la position (x,y) par rapport au conteneur dans lequel il est inclus et d'indiquer sa largeur et sa hauteur.

Toutefois, les Layout Manager constituent un des facteurs importants de la portabilité des interfaces graphiques notamment en gérant la disposition et le placement des composants après redimentionnement du conteneur.

11.2. Le positionnement des composants

Lorsqu'on intègre un composant graphique dans un conteneur, il n'est pas nécessaire de préciser son emplacement car il est déterminé de façon automatique : la mise en forme est dynamique. On peut influencer cette mise en page en utilisant un gestionnaire de mise en page (Layout Manager) qui définit la position de chaque composant inséré. Dans ce cas, la position spécifiée est relative par rapport aux autres composants.

Chaque layout manager implémente l'interface java.awt.LayoutManager.

Il est possible d'utiliser plusieurs gestionnaires de mise en forme pour définir la présentation des composants. Par défaut, c'est la classe FlowLayout qui est utilisée pour la classe Panel et la classe BorderLayout pour Frame et Dialog.

Pour affecter une nouvelle mise en page, il faut utiliser la méthode setLayout() de la classe Container.

Exemple (code java 1.1) :

```
Panel p = new Panel();
FlowLayout fl = new GridLayout(5,5);
p.setLayout(fl);

// ou p.setLayout( new GridLayout(5,5));
```

Les layout manager ont 3 avantages :

- l'aménagement des composants graphiques est délégué aux layout manager (il est inutile d'utiliser les coordonnées absolues)
- en cas de redimensionnement de la fenêtre, les contrôles sont automatiquement agrandis ou réduits
- ils permettent une indépendance vis à vis des plateformes.

Pour créer un espace entre les composants et le bord de leur conteneur, il faut rédifinir la méthode getInsets() d'un conteneur : cette méthode est héritée de la classe Container.

Exemple (code java 1.1) :

```
public Insets getInsets() {
   Insets normal = super.getInsets();
   return new Insets(normal.top + 10, normal.left + 10,
   normal.bottom + 10, normal.right + 10);
}
```

Cette Exemple permet de laisser 10 pixels en plus entre chaque bords du conteneur.

11.2.1. La mise en page par flot (FlowLayout)

La classe FlowLayout (mise en page flot) place les composants ligne par ligne de gauche à droite. Chaque ligne est complétée progressivement jusqu'à être remplie, puis passe à la suivante. Chaque ligne est centrée par défaut. C'est la mise en page par défaut des applets.

Il existe plusieurs constructeurs :

Constructeur	Role
FlowLayout( );	
FlowLayout(int align);	Permet de préciser l'alignement des composants dans le conteneur (CENTER, LEFT, RIGHT …). Par défaut, align vaut CENTER
FlowLayout(int, int hgap, int vgap);	Permet de préciser et l'alignement horizontal et vertival dont la valeur par défaut est 5.

Exemple (code java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      setLayout(new FlowLayout());
      add(new Button("Bouton 1"));
      add(new Button("Bouton 2"));
      add(new Button("Bouton 3"));

      pack();

      show(); // affiche la fenetre

   }

   public static void main(String[] args) {

      new MaFrame();

   }

}
```

Chaque applet possède une mise en page flot implicitement initialisée à FlowLayout(FloawLayout.CENTER,5,5).

FlowLayout utilise les dimensions de son conteneur comme seul prinicipe de mise en forme des composants. Si les dimensions du conteneurs changent, le positionnement des composants est recalculé.

Exemple : la fenêtre précédente est simplement redimensionnée

11.2.2. La mise en page bordure (BorderLayout)

Avec ce Layout Manager, la disposition des composants est commandée par une mise en page en bordure qui découpe la surface en cinq zones : North, South, East, West, Center. On peut librement utiliser une ou plusieurs zones.

BorderLayout consacre tout l'espace du conteneur aux composants. Le composant du milieu dispose de la place inutilisée par les autres composants.

Il existe plusieurrs constructeurs :

Constructeur	Role
BorderLayout( )	
BorderLayout (int hgap,int vgap)	Permet de préciser l'espacement horizontal et vertical des composants.

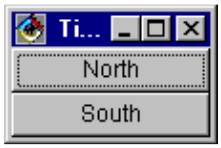
Exemple (code java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle("
      Titre de la Fenetre ");
      setSize(300, 150);
      setLayout(new
      BorderLayout());
      add("North", new Button(" bouton haut "));
      add("South", new Button(" bouton bas "));
      add("West",  new Button(" bouton gauche "));
      add("East",  new Button(" bouton droite "));
      add("Center", new Button(" bouton milieu "));
      pack();
      show(); // affiche la fenetre
   }

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

Il est possible d'utiliser deux méthodes add surchargées de la classe Container : add(String, Component) ou le premier paramètre précise l'orientation du composants ou add(Component, Objet) ou le second paramètre précise la position sous forme de constante définie dans la classe BorderLayout.

Exemple (code java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {

      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      setLayout(new BorderLayout());
      add(new Button("North"), BorderLayout.NORTH);
      add(new Button("South"), BorderLayout.SOUTH);
      pack();
      show(); // affiche la fenetre

   }

   public static void main(String[] args) {

      new MaFrame();

   }

}
```



11.2.3. La mise en page de type carte (CardLayout)

Aide à construire des boîtes de dialogue composée de plusieurs onglets. Un onglet se compose généralement de plusieurs contrôles : on insère des panneaux dans la fenêtre utilisée par le CardLayout Manager. Chaque panneau correspond à un onglet de boîte de dialogue et contient plusieurs contrôles. Par défaut, c'est le premier onglet qui est affiché.

Ce layout possède deux constructeurs :

Constructeurs	Role
CardLayout()	
CardLayout(int, int)	Permet de préciser l'espace horizontal et vertical du tour du composant

Exemple (code java 1.1) :

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

```
public class MaFrame extends Frame {

   public MaFrame() {

      super();
      setTitle("Titre de la Fenetre ");
      setSize(300,150);
      CardLayout cl = new CardLayout();
      setLayout(cl);

      //création d'un panneau contenant les controles d'un onglet
      Panel p = new Panel();

      //ajouter les composants au panel
      p.add(new Button("Bouton 1 panneau 1"));
      p.add(new Button("Bouton 2 panneau 1"));

      //inclure le panneau dans la fentre sous le nom "Page1"
      // ce nom est utilisé par show()

      add("Page1",p);

      //déclaration et insertion de l'onglet suivant
      p = new Panel();
      p.add(new Button("Bouton 1 panneau 2"));
      add("Page2", p);

      // affiche la fenetre
      pack();
      show();

   }

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

}
```

Lors de l'insertion d'un onglet, un nom doit lui être attribué. Les fonctions nécessaires pour afficher un onglet de boîte de dialogue ne sont pas fournies par les méthodes du conteneur, mais seulement par le Layout Manager. Il est nécessaire de sauvegarder temporairement le Layout Manager dans une variable ou déterminer le gestionnaire en cours par un appel à getLayout(). Pour appeler un onglet donné, il faut utiliser la méthode show() du CardLayout Manager.

Exemple (code java 1.1) :

```
((CardLayout)getLayout()).show(this, "Page2");
```

Les méthodes first(), last(), next() et previous() servent à parcourir les onglets de boîte de dialogue :

Exemple (code java 1.1) :

```
((CardLayout)getLayout()).first(this);
```

11.2.4. La mise en page GridLayout

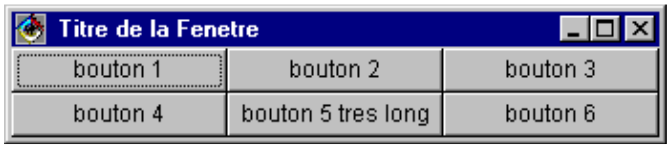
Ce Layout Manager établit un réseau de cellule identiques qui forment une sorte de quadrillage invisible : les composants sont organisés en lignes et en colonnes. Les éléments insérés dans la grille ont tous la même taille. Les cellules du quadrillage se remplissent de droite à gauche ou de haut en bas.

Il existe plusieurs constructeurs :

Constructeur	Role
GridLayout(int, int);	Les deux premiers entiers spécifient le nombre de lignes ou de colonnes de la grille.
GridLayout(int, int, int, int);	permet de préciser en plus l'espacement horizontal et vertical des composants.

Exemple (code java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      setLayout(new GridLayout(2, 3));
      add(new Button("bouton 1"));
      add(new Button("bouton 2"));
      add(new Button("bouton 3"));
      add(new Button("bouton 4"));
      add(new Button("bouton 5 tres long"));
      add(new Button("bouton 6"));

      pack();

      show(); // affiche la fenetre

   }

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

}
```

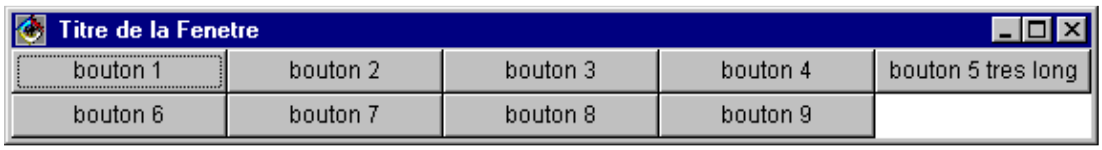


Attention : lorsque le nombre de ligne et de colonne est spécifié alors le nombre de colonne est ignoré. Ainsi par Exemple GridLayout(5,4) est équivalent à GridLayout(5,0).

Exemple (code java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
```

```
      setSize(300, 150);
      setLayout(new GridLayout(2, 3));
      add(new Button("bouton 1"));
      add(new Button("bouton 2"));
      add(new Button("bouton 3"));
      add(new Button("bouton 4"));
      add(new Button("bouton 5 tres long"));
      add(new Button("bouton 6"));
      add(new Button("bouton 7"));
      add(new Button("bouton 8"));
      add(new Button("bouton 9"));

      pack();
      show(); // affiche la fenetre

   }

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

}
```



11.2.5. La mise en page GridBagLayout

Ce gestionnaire (grille étendue) est le plus riche en fonctionnalités : le conteneur est divisé en cellules égales mais un composants peut occuper plusieurs cellules de la grille et il est possible de faire une distribution dans des cellules distinctes. Un objet de la classe GridBagConstraints permet de donner les indications de positionnement et de dimension à l'objet GridBagLayout.

Les lignes et les colonnes prennent naissance au moment ou les controles sont ajoutés. Chaque controle est associé à un objet de la classe GridBagConstraints qui indique l'emplacement voulu pour le contrôle.

Exemple (code java 1.1) :

```
GridBagLayout gb1 = new GridBagLayout( );
GridBagConstraints gbc = new GridBagConstraints( );
```

Les variables d'instances pour manipuler l'objet GridBagLayoutConstrainst sont :

Variable	Role
gridx et gridy	Ces variables contiennent les coordonnées de l'origine de la grille. Elles permettent un positionnement précis à une certaine position d'un composant. Par défaut elles ont la valeur GrigBagConstraint.RELATIVE qui indique qu'un composant se range à droite du précédent
gridwidth, gridheight	Définissent combien de cellules va occuper le composant (en hauteur et largeur). Par défaut la valeur est 1. L'indication est relative aux autres composants de la ligne ou de la colonne. La valeur GridBagConstraints.REMAINDER spécifie que le prochain composant inséré sera le dernier de la ligne ou de la colonne courante. La valeur GridBagConstraints.RELATIVE place le composant après le dernier composant d'une ligne ou d'une colonne.

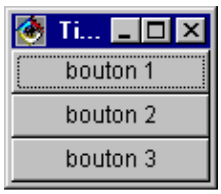
fill	Définit le sort d'un composant plus petit que la cellule de la grille. GridBagConstraints.NONE conserve la taille d'origine : valeur par défaut GridBagConstraints.HORIZONTAL dilaté horizontalement GridBagConstraints.VERTICAL dilaté verticalement GridBagConstraints.BOTH dilatés aux dimensions de la cellule
ipadx, ipady	Permettent de définir l'agrandissement horizontal et vertical des composants. Ne fonctionne que si une dilatation est demandée par fill. La valeur par défaut est (0,0).
anchor	Lorsqu'un composant est plus petit que la cellule dans laquelle il est inséré, il peut être positionné à l'aide de cette variable pour définir le côté par lequel le controle doit être aligné dans la cellule. Les variables possibles sont NORTH, NORTHWEST, NORTHEAST, SOUTH, SOUTHWEST, SOUTHEAST, WEST et EAST
weightx, weighty	Permettent de définir la répartition de l'espace en cas de changement de dimension

Exemple (code java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);

      Button b1 = new Button(" bouton 1 ");
      Button b2 = new Button(" bouton 2 ");
      Button b3 = new Button(" bouton 3 ");

      GridBagLayout gb = new GridBagLayout();

      GridBagConstraints gbc = new GridBagConstraints();
      setLayout(gb);

      gbc.fill = GridBagConstraints.BOTH;
      gbc.weightx = 1;
      gbc.weighty = 1;
      gb.setConstraints(b1, gbc); // mise en forme des objets
      gb.setConstraints(b2, gbc);
      gb.setConstraints(b3, gbc);

      add(b1);
      add(b2);
      add(b3);

      pack();

      show(); // affiche la fenetre

   }

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

}
```

Cette Exemple place trois boutons l'un à coté de l'autre. Ceci permet en cas de changement de dimension du conteneur de conserver la mise en page : la taille des composants est automatiquement ajustée.

Pour placer les 3 boutons l'un au dessus de l'autre, il faut affecter la valeur 1 à la variable gbc.gridx.



11.3. La création de nouveaux composants à partir de Panel

Il est possible de définir de nouveau composant qui hérite directement de Panel

Exemple (code java 1.1) :

```
class PanneauClavier extends Panel {

   PanneauClavier()
   {
      setLayout(new GridLayout(4,3));

      for (int num=1; num <= 9 ; num++) add(new Button(Integer.toString(num)));
      add(new Button(«*»);
      add(new Button(«0»);
      add(new Button(«# »);

   }

}

public class demo exteds Applet {

   public void init() { add(new PanneauClavier()); }

}
```

11.4. Activer ou desactiver des composants

L'activation ou la désactivation d'un composant se fait grace à sa méthode setEnabled(boolean). La valeur booléenne passée en paramètres indique l'état du composant (false : interdit l'usage du composant). Cette méthode est un moyen d'interdire à un composant d'envoyer des évenements utilisateurs.

11.5. Afficher une image dans une application.

L'image doit préalablement être charger grace à la classe Toolkit.

Cette section est en cours d'écriture

12. L'interception des actions de l'utilisateur

N'importe quel interface graphique doit interagir avec l'utilisateur et donc réagir a certains événements. Le modèle de gestion de ces évenements à changer entre le JDK 1.0 et 1.1.

Ce chapitre traite de la capture des ces événements pour leur associer des traitements. Il contient plusieurs sections :

12.1. Intercepter les actions de l'utilisateur avec Java version 1.0

Cette section est en cours d'écriture

12.2. Intercepter les actions de l'utilisateur avec Java version 1.1

Les événements utilisateurs sont gérés par plusieurs interfaces EventListener.

Les interfaces EventListener permettent à un composants de générer des événements utilisateurs. Une classe doit contenir une interface auditeur pour chaque type de composant :

- ActionListener : clic de souris ou enfoncement de la touche Enter
- ItemListener : utilisation d'une liste ou d'une case à cocher
- MouseMotionListener : evénément de souris
- WindowListener : événement de fenetre

L'ajout d'une interface EventListener impose plusieurs ajouts dans le code :

1. importer le groupe de classe java.awt.event

Exemple (code java 1.1) :

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

2. la classe doit déclarer qu'elle utilisera une ou plusieurs interfaces d'écoute

Exemple (code java 1.1) :

```
public class AppletAction extends Applet implements ActionListener{
```

Pour déclarer plusieurs interfaces, il suffit de les séparer par des virgules

Exemple (code java 1.1) :

```
public class MonApplet extends Applet implements ActionListener, MouseListener {
```

3. Appel à la méthode addXXX() pour enregistrer l'objet qui gerera les évenements XXX du composant

Il faut configurer le composant pour qu'il possède un "écouteur" pour l'événement utilisateur concerné.

Exemple (code java 1.1) : création d'un bouton capable de réagir à un evénements

```
Button b = new Button(«boutton»);
b.addActionListener(this);
```

Ce code créé l'objet de la classe Button et appelle sa méthode addActionListener(). Cette méthode permet de préciser qu'elle sera la classe qui va gérer l'évenement utilisateur de type ActionListener du bouton. Cette classe doit impérativement implémenter l'interface de type EventListener correspondante soit dans cette exemple ActionListener. L'instruction this indique que la classe elle même recevra et gérera l'évenement utilisateur.

L'apparition d'un évenement utilisateur généré par un composant doté d'un auditeur appelle automatiquement une méthode, qui doit se trouver dans la classe référencée dans l'instruction qui lie l'auditeur au composant. Dans l'exemple, cette méthode doit être située dans la même classe parce que c'est l'objet lui même qui est spécifié avec l'instruction this. Une autre classe indépendante peut être utilisée : dans ce cas il faut préciser une instance de cette classe en temps que paramètre.

4. implémenter les méthodes déclarées dans les interfaces

Chaque auditeur possède des méthodes différentes qui sont appelées pour traiter leurs évenéments. Par exemple, l'interface ActionListener envoie des évenements à une classe nommée actionPerformed( ).

Exemple (code java 1.1) :

```
public void actionPerformed(ActionEvent evt) {
   //insérer ici le code de la méthode
};
```

Pour identifier le composant qui a généré l'evénement il faut utiliser la méthode getActionCommand() de l'objet ActionEvent fourni en paramètre de la méthode :

Exemple (code java 1.1) :

```
String composant = evt.getActionCommand();
```

getActionCommand renvoie une chaine de caractères. Si le composant est un bouton, alors il renvoie le texte du bouton, si le composant est une zone de saisie, c'est le texte saisie qui sera renvoyé (il faut appuyer sur "Entrer" pour générer l'événement), etc …

La méthode getSource() renvoie l'objet qui a généré l'événement. Cette méthode est plus sure que la précédente

Exemple (code java 1.1) :

```
Button b = new Button(« bouton »);

…

void public actionPerformed(actionEvent evt) {
   Object source = evt.getSource();

   if (source == b) // action a effectuer
}
```

La méthode getSource() peut être utilisé avec tous les évenements utilisateur.

Exemple (code java 1.1) : Exemple complet qui affiche le composant qui a généré l'événement

```
package applets;

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

public class AppletAction extends Applet implements ActionListener{

   public void actionPerformed(ActionEvent evt) {
      String composant = evt.getActionCommand();
      showStatus("Action sur le composant : " + composant);
   }


   public void init() {
      super.init();

      Button b1 = new Button("boutton 1");
      b1.addActionListener(this);
      add(b1);

      Button b2 = new Button("boutton 2");
      b2.addActionListener(this);
      add(b2);

      Button b3 = new Button("boutton 3");
      b3.addActionListener(this);
      add(b3);
   }
}
```

12.2.1. L'interface ItemListener

Cette interface permet de réagir à la sélection de cases à cocher et de liste d'options. Pour qu'un composant genère des évenements, il faut utiliser la méthode addItemListener().

Exemple (code java 1.1) :

```
Checkbox cb = new Checkbox(« choix »,true);
cb.addItemListener(this);
```

Ces évenements sont reçus par la méthode itemStateChanged() qui attend un objet de type ItemEvent en argument

Pour déterminer si une case à cocher est sélectionnée ou inactive, utiliser la méthode getStateChange() avec les constantes ItemEvent.SELECTED ou ItemEvent.DESELECTED.

Exemple (code java 1.1) :

```
package applets;

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

public class AppletItem extends Applet implements ItemListener{

   public void init() {
      super.init();
      Checkbox cb = new Checkbox("choix 1", true);
      cb.addItemListener(this);
      add(cb);
   }

   public void itemStateChanged(ItemEvent item) {
      int status = item.getStateChange();
      if (status == ItemEvent.SELECTED)
         showStatus("choix selectionne");
      else
         showStatus("choix non selectionne");
   }

}
```

Pour connaitre l'objet qui a généré l'événement, il faut utiliser la méthode getItem().

Pour déterminer la valeur sélectionnée dans une combo box, il faut utiliser la méthode getItem() et convertir la valeur en chaine de caractères.

Exemple (code java 1.1) :

```
package applets;

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

public class AppletItem extends Applet implements ItemListener{

   public void init() {
      Choice c = new Choice();
      c.add("choix 1");
      c.add("choix 2");
      c.add("choix 3");
      c.addItemListener(this);
      add(c);
}

   public void itemStateChanged(ItemEvent item) {
      Object obj = item.getItem();
      String selection = (String)obj;
      showStatus("choix : "+selection);
   }
}
```

12.2.2. L'interface TextListener

Cette interface permet de réagir au modification de zone de saisie ou de texte.

La méthode addTextListener() permet à un composant de texte de générer des événements utilisateur. La méthode TextValueChanged() reçoit les évenements.

Exemple (code java 1.1) :

```
package applets;

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

public class AppletText extends Applet implements TextListener{

   public void init() {
      super.init();

      TextField t = new TextField("");
      t.addTextListener(this);
      add(t);
   }

   public void textValueChanged(TextEvent txt) {
      Object source = txt.getSource();
      showStatus("saisi = "+((TextField)source).getText());
   }
}
```

12.2.3. L'interface MouseMotionListener

La méthode addMouseMotionListener() permet de gérer les évenements liés à des mouvements de souris. La méthode mouseDragged() et mouseMoved() reçoivent les évenements.

Exemple (code java 1.1) :

```
package applets;

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

public class AppletMotion extends Applet implements MouseMotionListener{
   private int x;
   private int y;

   public void init() {
      super.init();
      this.addMouseMotionListener(this);
   }

   public void mouseDragged(java.awt.event.MouseEvent e) {}

   public void mouseMoved(MouseEvent e) {
      x = e.getX();
      y = e.getY();
      repaint();
      showStatus("x = "+x+" ; y = "+y);
   }

   public void paint(Graphics g) {
      super.paint(g);
      g.drawString("x = "+x+" ; y = "+y,20,20);
   }
}
```

12.2.4. L'interface MouseListener

Cette interface permet de réagir aux clics de souris. Les méthodes de cette interface sont :

- public void mouseClicked(MouseEvent e);
- public void mousePressed(MouseEvent e);
- public void mouseReleased(MouseEvent e);
- public void mouseEntered(MouseEvent e);
- public void mouseExited(MouseEvent e);

Exemple (code java 1.1) :

```
package applets;

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

public class AppletMouse extends Applet implements MouseListener {
   int nbClick = 0;

   public void init() {
      super.init();
      addMouseListener(this);
   }

   public void mouseClicked(MouseEvent e) {
      nbClick++;
      repaint();
   }

   public void mouseEntered(MouseEvent e) {}

   public void mouseExited(MouseEvent e) {}

   public void mousePressed(MouseEvent e) {}

   public void mouseReleased(MouseEvent e) {}

   public void paint(Graphics g) {
      super.paint(g);
      g.drawString("Nombre de clics : "+nbClick,10,10);
   }
}
```

Une classe qui implémente cette interface doit définir ces 5 méthodes. Si toutes les méthodes ne doivent pas être utiliser, il est possible de définir une classe qui hérite de MouseAdapter. Cette classe fournit une implémentation par défaut de l'interface MouseListener.

Exemple (code java 1.1) :

```
class gestionClics extends MouseAdapter {

   public void mousePressed(MouseEvent e) {
      //traitement
   }
}
```

Dans le cas d'une classe qui hérite d'une classe Adapter, il suffit de redéfinir la ou les méthodes qui contiendront du code pour traiter les événements concernés. Par défaut, les différentes méthodes définies dans l'Adapter ne font rien.

Cette nouvelle classe ainsi définie doit etre passée en paramètre à la méthode addMouseListener() au lieu de this qui indiquait que la classe répondait elle même au événement.

12.2.5. L'interface WindowListener

La méthode addWindwowListener() permet à un objet Frame de générer des événements. Les méthodes de cette interface sont :

- public void windowOpened(WindowEvent e)
- public void windowClosing(WindowEvent e)
- public void windowClosed(WindowEvent e)
- public void windowIconified(WindowEvent e)
- public void windowDeinconified(WindowEvent e)
- public void windowActivated(WindowEvent e)
- public void windowDeactivated(WindowEvent e)

windowClosing est appelée lorque l'on clique sur la case système de fermeture de fenetre. windowClosed est appelé après la fermeture de la fenetre : cette méthode n'est utile que si la fermeture de la fenetre n'entraine pas la fin de l'application.

Exemple (code java 1.1) :

```
package test;

import java.awt.event.*;

class GestionnaireFenetre extends WindowAdpter {

   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
}
```

Exemple (code java 1.1) :

```
package test;

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

public class TestFrame extends Frame {

   private GestionnaireFenetre gf = new GestionnaireFenetre();


   public TestFrame(String title) {
     super(title);
     addWindowListener(gf);
   }

   public static void main(java.lang.String[] args) {
      try {
         TestFrame tf = new TestFrame("TestFrame");
         tf.setVisible(true);
      } catch (Throwable e) {
         System.err.println("Erreur");
         e.printStackTrace(System.out);
      }
   }
}
```

12.2.6. Les différentes implémentations des Listener

La mise en oeuvre des Listeners peut se faire selon différentes formes : la classe implémentant elle même l'interface, une classe indépendante, une classe interne, une classe interne anonyme.

12.2.6.1. Une classe implémentant elle même le listener

Exemple (code java 1.1) :

```
package test;

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

public class TestFrame3 extends Frame implements WindowListener {

   public TestFrame3(String title) {
      super(title);
      this.addWindowListener(this);
   }

   public static void main(java.lang.String[] args) {
      try {
         TestFrame3 tf = new TestFrame3("testFrame3");
         tf.setVisible(true);
      } catch (Throwable e) {
         System.err.println("Erreur");
         e.printStackTrace(System.out);
      }
   }

   public void windowActivated(java.awt.event.WindowEvent e) {}

   public void windowClosed(java.awt.event.WindowEvent e) {}

   public void windowClosing(java.awt.event.WindowEvent e) {
      System.exit(0);
   }

   public void windowDeactivated(java.awt.event.WindowEvent e) {}

   public void windowDeiconified(java.awt.event.WindowEvent e) {}

   public void windowIconified(java.awt.event.WindowEvent e) {}

   public void windowOpened(java.awt.event.WindowEvent e) {}

}
```

12.2.6.2. Une classe indépendante implémentant le listener

Exemple (code java 1.1) :

```
package test;

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

public class TestFrame4 extends Frame {

   public TestFrame4(String title) {
      super(title);
      gestEvt ge = new gestEvt();
      addWindowListener(ge);
   }

   public static void main(java.lang.String[] args) {
      try {
         TestFrame4 tf = new TestFrame4("testFrame4");
         tf.setVisible(true);
      } catch (Throwable e) {
         System.err.println("Erreur");
```

```
         e.printStackTrace(System.out);
      }
   }
}
```

Exemple (code java 1.1) :

```
package test;

import java.awt.event.*;

public class gestEvt implements WindowListener {

   public void windowActivated(WindowEvent e) {}
   public void windowClosed(WindowEvent e) {}
   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
   public void windowDeactivated(WindowEvent e) {}
   public void windowDeiconified(WindowEvent e) {}
   public void windowIconified(WindowEvent e) {}
   public void windowOpened(WindowEvent e) {}
}
```

12.2.6.3. Une classe interne

Exemple (code java 1.1) :

```
package test;

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

public class TestFrame2 extends Frame {

   class gestEvt implements WindowListener {
      public void windowActivated(WindowEvent e) {};
      public void windowClosed(WindowEvent e) {};
      public void windowClosing(WindowEvent e) {
         System.exit(0);
      };
      public void windowDeactivated(WindowEvent e) {};
      public void windowDeiconified(WindowEvent e) {};
      public void windowIconified(WindowEvent e) {};
      public void windowOpened(WindowEvent e) {};
   };

   private gestEvt ge = new TestFrame2.gestEvt();


   public TestFrame2(String title) {
     super(title);
     addWindowListener(ge);
   }

   public static void main(java.lang.String[] args) {
      try {
         TestFrame2 tf = new TestFrame2("TestFrame2");
         tf.setVisible(true);
      } catch (Throwable e) {
         System.err.println("Erreur");
         e.printStackTrace(System.out);
      }
   }
}
```

12.2.6.4. une classe interne anonyme

Exemple (code java 1.1) :

```
package test;

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

public class TestFrame1 extends Frame {

   public TestFrame1(String title) {
      super(title);
      addWindowListener(new WindowAdapter() {
         public void windowClosed(.WindowEvent e) {
            System.exit(0);
         };
      });
   }

   public static void main(java.lang.String[] args) {
      try {
         TestFrame1 tf = new TestFrame1("TestFrame");
         tf.setVisible(true);
      } catch (Throwable e) {
         System.err.println("Erreur");
         e.printStackTrace(System.out);
      }
   }
}
```

12.2.7. Résumé

Le mécanisme mis en place pour intercepter des événements est le même quel que soit ces événements :

- associer au composant qui est à l'origine de l'évenement un controleur adéquat : utilisation des méthodes addXXXListener() Le paramètre de ces méthodes indique l'objet qui a la charge de répondre au message : cet objet doit implémenter l'interface XXXListener correspondant ou dérivé d'une classe XXXAdapter (créer une classe qui implémente l'interface associé à l'événement que l'on veut gérer. Cette classe peut être celle du composant qui est à l'origine de l'événement (facilité d'implémentation) ou une classe indépendante qui détermine la frontière entre l'interface graphique (émission d'événement) et celle qui représente la logique de l'application (traitement des événements)).
- les classes XXXAdapter sont utiles pour créer des classes dédiées au traitement des évenements car elles implémentent des méthodes par défaut pour celles définies dans l'interface XXXListener dérivées de EventListener. Il n'existe une classe Adapter que pour les interface qui possédent plusieurs méthodes.
- implémenter la méthode associé à l'événement qui fournit en paramètre un objet de type AWTEvent (classe mère de tout événement) qui contient des informations utiles (position du curseur, état du clavier ...).

13. Le développement d'interface graphique avec SWING

Swing fait partie de la bibliothèque Java Foundation Classes. C'est une extension de l'AWT qui a été intégrée au JDK depuis sa version 1.2. Cette bibliothèque existe séparement pour le JDK 1.1.

La bibliothèque JFC contient :

- l'API Swing : de nouvelles classes et interfaces pour construire des interfaces graphiques
- Accessibility API :
- 2D API: support de graphisme en 2D
- API pour l'impression et le cliquer−glisser

13.1. Présentation de Swing

Swing propose de nombreux composants dont certains possède des fonctions étendues, une utilisation des mécanismes de gestion d'événements performants (ceux introduits par le JDK 1.1) et une apparence modifiable à la volée (une interface qui emploie le style du système d'exploitation Windows ou Motif ou un nouveau style spécifique à Java nommé Metal).

Tous les éléments de Swing font partie d'un package qui a changé plusieurs fois de nom : le nom du package dépend de la version du J.D.K. utilisé :

- com.sun.java.swing : jusqu'à la version 1.1 beta 2 de Swing des JFC 1.1 et J.D.K. 1.2 beta 4
- java.awt.swing : utilisé par le J.D.K. 1.2 beta 2 et 3
- javax.swing : à partir des versions de Swing 1.1 beta 3 et J.D.K. 1.2 RC1

Les composants Swing forment un nouvelle hériarchie parallèle à celle de l'AWT. L'ancètre de cette hiérarchie est le composant JComponent. Presque tous ces composants sont écrits en pure Java : ils ne possèdent aucune partie native sauf ceux qui assurent l'interface avec le système d'exploitation : JApplet, JDialog, JFrame, et JWindow. Cela permet aux composants de toujours avoir la même apparence quelque soit le système.

Tous les composants Swing possèdent les caractèristiques suivantes :

- ce sont des beans
- ce sont des composants légers (pas de partie native) hormis quelques exceptions.
- leurs bords peuvent être changés

La procédure à suivre pour utiliser un composant Swing est identique à celle des composants AWT : créer le composant en appelant son constructeur, appeler les méthodes du composant si nécessaire pour le personnaliser et l'ajouter dans un conteneur.

Swing utilise la même infrastructure de classes que AWT, ce qui permet de mélanger des composants Swing et AWT dans la même interface. Sun recommande toutefois d'éviter de les mélanger car certains peuvent ne pas être restitués correctement.

Les composants Swing utilisent des modèles pour contenir leurs états ou leur données. Ces modèles sont des classes

particulières qui possèdent toutes un comportement par défaut.

13.2. Les packages Swing

Swing contient plusieurs packages :

javax.swing	package principal : il contient les interfaces, les principaux composants, les modèles par défaut
javax.swing.border	Classes représentant les bordures
javax.swing.colorchooser	Classes définissant un composant pour la sélection de couleurs
javax.swing.event	Classes et interfaces pour les événements spécifique à Swing. Les autres événements sont ceux de AWT (java.awt.event)
javax.swing.filechooser	Classes définissant un composant pour la sélection de fichiers
javax.swing.plaf	Classes et interfaces génériques pour gérer l'apparence
javax.swing.plaf.basic	Classes et interfaces de base pour gérer l'apparence
javax.swing.plaf.metal	Classes et interfaces pour définir l'apparence Metal qui est l'apparence par défaut
javax.swing.table	Classes définissant un composant pour la présentation de données sous forme de tableau
javax.swing.text	Classes et interfaces de bases pour les composants manipulant du texte
javax.swing.text.html	Classes permettant le support du format HTML
javax.swing.text.html.parser	Classes permettant d'analyser des données au format HTML
javax.swing.text.rtf	Classes permettant le support du format RTF
javax.swing.tree	Classes définissant un composant pour la présentation de données sous forme d'arbre
javax.swing.undo	Classes permettant d'implémenter les fonctions annuler/refaire

13.3. Un exemple de fenêtre autonome

La classe de base d'une application est la classe JFrame. Son rôle est équivalent à la classe Frame de l'AWT et elle s'utilise de la même façon

Exemple (code java 1.1) :

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

public class swing1 extends JFrame {

   public swing1() {
      super("titre de l'application");

      WindowListener l = new WindowAdapter() {
         public void windowClosing(WindowEvent e){
            System.exit(0);
         }
      };

      addWindowListener(l);
      setSize(200,100);
```

```
      setVisible(true);
   }

   public static void main(String [] args){
      JFrame frame = new swing1();
   }
}
```

13.4. Les composants Swing

Il existe des composants Swing équivalents pour chacun des composants AWT avec des constructeurs semblables. De nombreux constructeurs acceptent comme argument un objet de type Icon, qui représente une petite image généralement stockée au format Gif.

Le constructeur d'un objet Icon admet comme seul paramètre le nom ou l'URL d'un fichier graphique

Exemple (code java 1.1) :

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

public class swing3 extends JFrame {

  public swing3() {

     super("titre de l'application");

      WindowListener l = new WindowAdapter() {
         public void windowClosing(WindowEvent e){
            System.exit(0);
         }
      };

      addWindowListener(l);

      ImageIcon img = new ImageIcon("tips.gif");
      JButton bouton = new JButton("Mon bouton",img);

      JPanel panneau = new JPanel();
      panneau.add(bouton);
      setContentPane(panneau);
      setSize(200,100);
      setVisible(true);
   }

  public static void main(String [] args){
      JFrame frame = new swing3();
  }
}
```

13.4.1. La classe JFrame

JFrame est l'équivalent de la classe Frame de l'AWT : les principales différences sont l'utilisation du double buffering qui améliore les rafraichissements et l'utilisation d'un panneau de contenu (contentPane) pour insérer des composants (ils ne sont plus insérer directement au JFrame mais à l'objet contentPane qui lui est associé). Elle représente une fenêtre principale qui possède un titre, une taille modifiable et éventuellement un menu.

La classe possède plusieurs constructeurs :

Constructeur	Rôle
JFrame()	

JFrame(String)	Création d’une instance en précisant le titre

Par défaut, la fenêtre créée n’est pas visible. La méthode setVisible() permet de l’afficher.

Exemple (code java 1.1) :

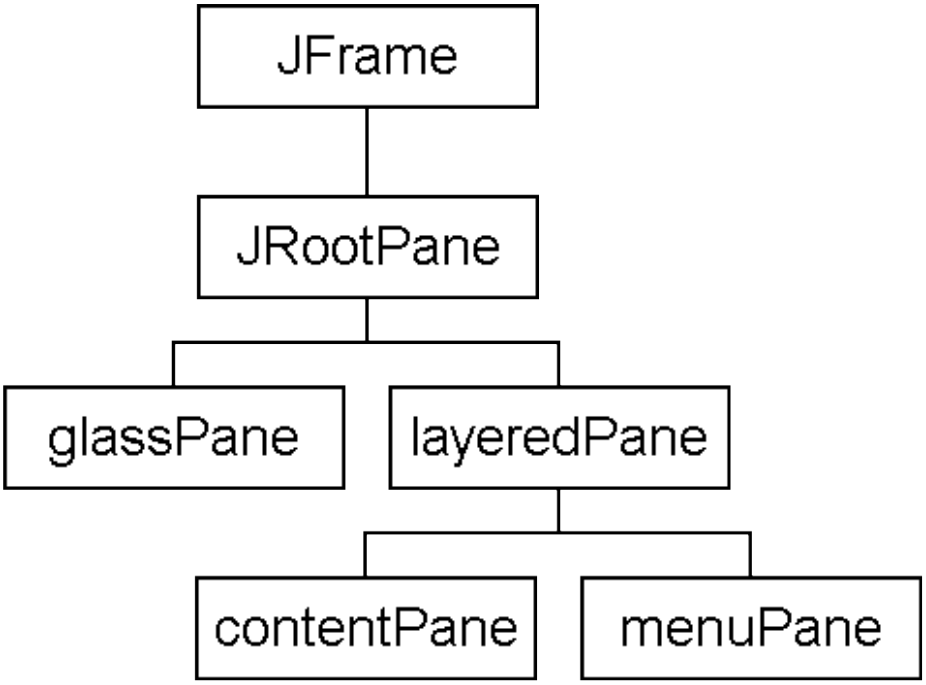
```
import javax.swing.*;

public class TestJFrame1 {

  public static void main(String argv[]) {
    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    f.setVisible(true);
  }
}
```

La gestion des événements est identique à celle utilisée dans l'AWT depuis le J.D.K. 1.1.

Exemple (code java 1.1) :

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

public class swing2 extends JFrame {

   public swing2() {

     super("titre de l'application");

      WindowListener l = new WindowAdapter() {
         public void windowClosing(WindowEvent e){
           System.exit(0);
         }
      };
     addWindowListener(l);

     JButton bouton = new JButton("Mon bouton");
     JPanel panneau = new JPanel();
     panneau.add(bouton);

     setContentPane(panneau);
     setSize(200,100);
     setVisible(true);
   }

   public static void main(String [] args){
      JFrame frame = new swing2();
   }
}
```

Tous les composants associés à un objet JFrame sont gérer par un objet de la classe JRootPane. Un objet JRootPane contient plusieurs Panes. Tous les composants ajoutés au JFame doivent être ajouté à un Pane du JRootPane et non au JFrame directement. C’est aussi à un de ces Panes qu’il faut associer un layout manager si nécessaire.



Le Pane le plus utilisé est le ContentPane. Le Layout manager par défaut du contentPane est BorderLayout. Il est possible de le changer :

Exemple (code java 1.1) :

```
...
f.getContentPane().setLayout(new FlowLayout());
...
```

Exemple (code java 1.1) :

```
import javax.swing.*;

public class TestJFrame2 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JButton b =new JButton("Mon bouton");
    f.getContentPane().add(b);
    f.setVisible(true);
  }
}
```

Le JRootPane se compose de plusieurs éléments :

- glassPane : un JPanel par défaut
- layeredPane qui se compose du contentPane (un JPanel par défaut) et du menuBar (un objet de type JMenuBar)

Le glassPane est un JPanel transparent qui se situe au dessus du layeredPane. Le glassPane peut être n'importe quel composant : pour le modifier il faut utiliser la méthode setGlassPane() en fournissant le composant en paramètre.

Le layeredPane regroupe le contentPane et le menuBar.

Le contentPane est par défaut un JPanel opaque dont le gestionnaire de présentation est un BorderLayout. Ce panel peut être remplacé par n'importe quel composant grâce à la méthode setContentPane().

Attention : il ne faut pas utiliser directement la méthode setLayout() d'un objet JFrame sinon une exception est levée.

Exemple (code java 1.1) :

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

public class TestJFrame7 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setLayout(new FlowLayout());
    f.setSize(300,100);
    f.setVisible(true);
  }
}
```

Résultat :

```
C:\swing\code>java TestJFrame7
Exception in thread "main" java.lang.Error: Do not use javax.swing.JFrame.setLay
out() use javax.swing.JFrame.getContentPane().setLayout() instead
        at javax.swing.JFrame.createRootPaneException(Unknown Source)
        at javax.swing.JFrame.setLayout(Unknown Source)
        at TestJFrame7.main(TestJFrame7.java:8)
```

Le menuBar permet d'attacher un menu à la JFrame. Par défaut, le menuBar est vide. La méthode setJMenuBar() permet d'affecter un menu à la JFrame.

Exemple (code java 1.1) : Création d'un menu très simple

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

public class TestJFrame6 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JButton b =new JButton("Mon bouton");
    f.getContentPane().add(b);

    JMenuBar menuBar = new JMenuBar();
    f.setJMenuBar(menuBar);

    JMenu menu = new JMenu("Fichier");
    menu.add(menuItem);
    menuBar.add(menu);

    f.setVisible(true);
  }
}
```

13.4.1.1. Le comportement par défaut à la fermeture

Il possible préciser comment un objet JFrame, JInternalFrame, ou Jdialog réagit à sa fermeture grâce à la méthode setDefaultCloseOperation(). Cette méthode attend en paramètre une valeur qui peut être :

Constante	Rôle
WindowConstants.DISPOSE_ON_CLOSE	détruit la fenêtre
WindowConstants.DO_NOTHING_ON_CLOSE	rend le bouton de fermeture inactif
WindowConstants.HIDE_ON_CLOSE	cache la fenêtre

Cette méthode ne permet pas d'associer d'autres traitements. Dans ce cas, il faut intercepter l'évenement et lui associer les traitements.

Exemple (code java 1.1) : la fenêtre disparaît lors de sa fermeture mais l'application ne se termine pas.

```
import javax.swing.*;

public class TestJFrame3 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JButton b =new JButton("Mon bouton");
    f.getContentPane().add(b);

    f.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

    f.setVisible(true);
  }
}
```

13.4.1.2. La personnalisation de l'icone

La méthode setIconImage() permet de modifier l'icône de la JFrame.

Exemple (code java 1.1) :

```
import javax.swing.*;

public class TestJFrame4 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JButton b =new JButton("Mon bouton");
    f.getContentPane().add(b);
    f.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

    ImageIcon image = new ImageIcon("book.gif");
    f.setIconImage(image.getImage());
    f.setVisible(true);

  }
}
```

Si l'image n'est pas trouvée, alors l'icône est vide. Si l'image est trop grande, elle est redimensionnée.

13.4.1.3. Centrer une JFrame à l'écran

Par défaut, une JFrame est affichée dans le coin supérieur gauche de l'écran. Pour la centrer dans l'écran, il faut procéder comme pour une Frame : déterminer la position de la Frame en fonction de sa dimension et de celle de l'écran et utiliser la méthode setLocation() pour affecter cette position.

Exemple (code java 1.1) :

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

public class TestJFrame5 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JButton b =new JButton("Mon bouton");
    f.getContentPane().add(b);

    f.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

    Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
    f.setLocation(dim.width/2 - f.getWidth()/2, dim.height/2 - f.getHeight()/2);

    f.setVisible(true);
  }
}
```

13.4.1.4. Les évenements associées à un JFrame

La gestion des évenements associés à un objet JFrame est identique à celle utilisée pour un objet de type Frame de AWT.

Exemple (code java 1.1) :

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

public class TestJFrame8 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          System.exit(0);
        }
      }
    );

  }
}
```

13.4.2. Les étiquettes : la classe JLabel

Le composant JLabel propose les mêmes fonctionnalités que les intitulés AWT mais ils peuvent en plus contenir des icônes .

Cette classe possède plusieurs constructeurs :

Constructeurs	Rôle
JLabel()	Création d'une instance sans texte ni image
`JLabel(Icon)`	Création d'une instance en précisant l'image
JLabel(Icon, int)	Création d'une instance en précisant l'image et l'alignement horizontal
JLabel(String)	Création d'une instance en précisant le texte
JLabel(String, Icon, int)	Création d'une instance en précisant le texte, l'image et l'alignement horizontal
JLabel(String, int)	Création d'une instance en précisant le texte et l'alignement horizontal

Le composant JLabel permet d'afficher un texte et/ou une icône en précisant leur alignement. L'icône doit être au format GIF et peut être une animation dans ce format.

Exemple (code java 1.1) :

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

public class TestJLabel1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(100,200);

    JPanel pannel = new JPanel();
    JLabel jLabel1 =new JLabel("Mon texte dans JLabel");
    pannel.add(jLabel1);

    ImageIcon icone = new ImageIcon("book.gif");
    JLabel jLabel2 =new JLabel(icone);
    pannel.add(jLabel2);

    JLabel jLabel3 =new JLabel("Mon texte",icone,SwingConstants.LEFT);
    pannel.add(jLabel3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

La classe JLabel définie plusieurs méthodes pour modifier l'apparence du composant :

Méthodes	Rôle
setText()	Permet d'initialiser ou de modifier le texte affiché
setOpaque()	Indiquer si le composant est transparent (paramètre false) ou opaque (true)
setBackground()	Indique la couleur de fond du composant (setOpaque doit être à true)
setFont()	Permet de présicer la police du texte
setForeGround()	Permet de préciser la couleur du texte
setHorizontalAlignment()	Permet de modifier l'alignement horizontal du texte et de l'icône
setVerticalAlignment()	Permet de modifier l'alignement vertical du texte et de l'icône

setHorizontalTextAlignment()	Permet de modifier l'alignement horizontal du texte uniquement
setVerticalTextAlignment()	Permet de modifier l'alignement vertical du texte uniquement Exemple : jLabel.setVerticalTextPosition(SwingConstants.TOP);
setIcon()	Permet d'assigner une icône
setDisabledIcon()	Permet d'assigner une icône dans un état désactivée

L'alignement vertical par défaut d'un JLabel est centré. L'alignement horizontal par défaut est soit à droite si il ne contient que du texte, soit centré si il contient un image avec ou sans texte. Pour modifier cette alignement, il suffit d'utiliser les méthodes ci dessus en utilisant des constantes en paramètres : SwingConstants.LEFT, SwingConstants.CENTER, SwingConstants.RIGHT, SwingConstants.TOP, SwingConstants.BOTTOM

Par défaut, un JLabel est transparent : son fond n'est pas dessiné. Pour le dessiner, il faut utiliser la méthode setOpaque() :

Exemple (code java 1.1) :

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

public class TestJLabel2 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");

    f.setSize(100,200);
    JPanel pannel = new JPanel();

    JLabel jLabel1 =new JLabel("Mon texte dans JLabel 1");
    jLabel1.setBackground(Color.red);
    pannel.add(jLabel1);

    JLabel jLabel2 =new JLabel("Mon texte dans JLabel 2");
    jLabel2.setBackground(Color.red);
    jLabel2.setOpaque(true);
    pannel.add(jLabel2);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

Dans l'exemple, les 2 JLabel ont le fond rouge demandé par la méthode setBackground(). Seul le deuxième affiche un fond rouge car il est rendu opaque avec la méthode setOpaque().

Il est possible d'associer un raccourci clavier au JLabel qui permet de donner le focus à un autre composant. La méthode setDisplayedMnemonic() permet de définir le raccourci clavier. Celui ci sera activé en utilisant la touche Alt avec le caractère fourni en paramètre. La méthode setLabelFor() permet d'associer le composant fourni en paramètre au raccourci.

Exemple (code java 1.1) :

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

public class TestJLabel3 {
```

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

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();

    JButton bouton = new JButton("saisir");
    pannel.add(bouton);

    JTextField jEdit = new JTextField("votre nom");

    JLabel jLabel1 =new JLabel("Nom : ");
    jLabel1.setBackground(Color.red);
    jLabel1.setDisplayedMnemonic('n');
    jLabel1.setLabelFor(jEdit);
    pannel.add(jLabel1);
    pannel.add(jEdit);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

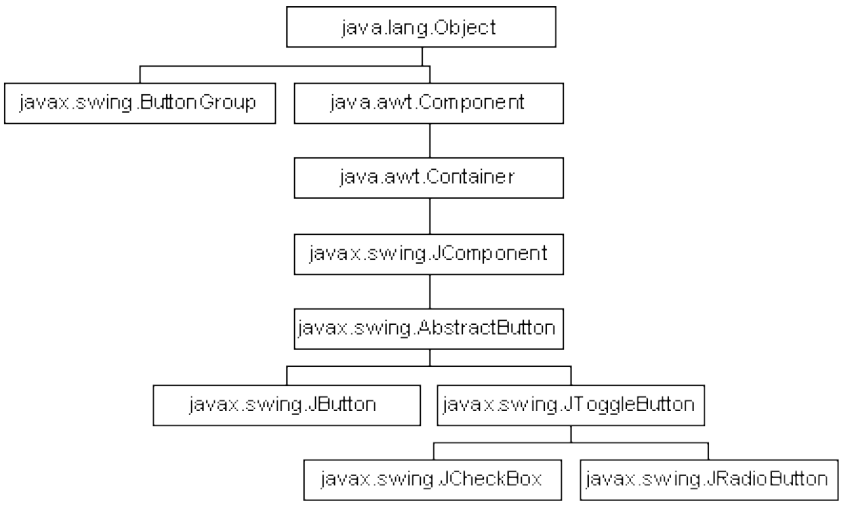
Dans l'exemple, à l'ouverture de la fenêtre, le focus est sur le bouton. Un appui sur Alt+'n' donne le focus au champ de saisie.

13.4.3 Les panneaux : la classe Jpanel

La classe JPanel est un conteneur utilisé pour regrouper et organiser des composants grâce à un gestionnaire de présentation (layout manager). Le gestionnaire par défaut d'un JPanel est un objet de la classe FlowLayout.

13.5. Les boutons

Il existe plusieurs boutons définis par Swing.



13.5.1. La classe AbstractButton

C'est une classe abstraite dont hérite les boutons Swing JButton, JMenuItem et JToggleButton.

Cette classe définie de nombreuses méthodes dont les principales sont :

Méthode	Rôle
AddActionListener	Associer un écouteur sur un évenement de type ActionEvent
AddChangeListener	Associer un écouteur sur un évenement de type ChangeEvent
AddItemListener	Associer un écouteur sur un évenement de type ItemEvent
doClick()	Déclencher un clic par programmation
getText()	Texte affiché par le composant
setDisabledIcon()	Associé une icone affichée lorsque le composant à l'état désélectionné
setDisabledSelectedIcon()	Associé une icone affichée lors du passage de la souris sur le composant à l'état désélectionné
setEnabled()	activer/désactiver le composant
setMnemonic()	associer un raccouci clavier
setPressedIcon()	Associé une icone affichée lorsque le composant est préssé
setRolloverIcon()	Associé une icone affichée lors du passage de la souris sur le composant
setRolloverSelectedIcon()	Associé une icone affichée lors du passage de la souris sur le composant à l'état sélectionné
setSelectedIcon()	Associé une icone affichée lorsque le composant à l'état sélectionné
setText()	Mise à jour du texte du composant
isSelected()	Indique si le composant est dans l'état sélectionné
setSelected()	Mise à jour de l'état selectionné du composant

Tous les boutons peuvent afficher du texte et/ou une image.

Il est possible de préciser une image différente lors du passage de la souris sur le composant et lors de l'enfoncement du bouton : dans ce cas, il faut créer trois images pour chacun des états (normal, enfoncé et survolé). L'image normale est associée au bouton grace au constructeur, l'image enfoncée grâce à la méthode setPressedIcon() et l'image lors d'un survole grâce à la méthode setRolloverIcon(). Il suffit enfin d'appeler la méthode setRolloverEnable() avec en paramètre la valeur true.

Exemple (code java 1.1) :

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

public class swing4 extends JFrame {

   public swing4() {
     super("titre de l'application");

     WindowListener l = new WindowAdapter() {
       public void windowClosing(WindowEvent e){
         System.exit(0);
       }
     };
     addWindowListener(l);

     ImageIcon imageNormale = new ImageIcon("arrow.gif");
     ImageIcon imagePassage =  new ImageIcon("arrowr.gif");
```

```
     ImageIcon imageEnfoncee = new ImageIcon("arrowy.gif");

     JButton bouton = new JButton("Mon bouton",imageNormale);
     bouton.setPressedIcon(imageEnfoncee);
     bouton.setRolloverIcon(imagePassage);
     bouton.setRolloverEnabled(true);
     getContentPane().add(bouton, "Center");

     JPanel panneau = new JPanel();
     panneau.add(bouton);
     setContentPane(panneau);
     setSize(200,100);
     setVisible(true);
   }

   public static void main(String [] args){
      JFrame frame = new swing4();
   }
}
```

Un bouton peut recevoir des évenements de type ActionEvents (le bouton a été activé), ChangeEvents, et ItemEvents.

Exemple (code java 1.1) : fermeture de l'application lors de l'activation du bouton

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

public class TestJButton3 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();

    JButton bouton1 = new JButton("Bouton1");
    bouton1.addActionListener( new ActionListener() {
       public void actionPerformed(ActionEvent e) {
         System.exit(0);
       }
    }
    );

    pannel.add(bouton1);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

Pour de plus amplements informations sur la gestion des événements, voir le chapitre correspondant.

13.5.2. La classe JButton : les boutons

JButton est un composant qui représente un bouton : il peut contenir un texte et/ou une icône.

Il ne gère pas d'état.

Les constructeurs sont :

Constructeur	Rôle
JButton()	

JButton(String)	préciser le texte du bouton
JButton(Icon)	préciser une icône
JButton(String, Icon)	préciser un texte et une icone

Il ne gère pas d'état. Toutes les indications concernant le contenu du composant JLabel sont valables pour le composant JButton.

Exemple (code java 1.1) : un bouton avec une image

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

public class swing3 extends JFrame {

   public swing3() {

      super("titre de l'application");

      WindowListener l = new WindowAdapter() {
         public void windowClosing(WindowEvent e){
            System.exit(0);
         }
      };
      addWindowListener(l);

      ImageIcon img = new ImageIcon("tips.gif");
      JButton bouton = new JButton("Mon bouton",img);

      JPanel panneau = new JPanel();
      panneau.add(bouton);
      setContentPane(panneau);
      setSize(200,100);
      setVisible(true);
  }

  public static void main(String [] args){
      JFrame frame = new swing3();
  }

}
```

L'image gif peut être une animation.

Dans un conteneur de type JRootPane, il est possible de définir un bouton par défaut grace à sa méthode setDefaultButton().

Exemple (code java 1.1) : définition d'un bouton par défaut dans un JFrame

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

public class TestJButton2 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();
    JButton bouton1 = new JButton("Bouton 1");
    pannel.add(bouton1);

    JButton bouton2 = new JButton("Bouton 2");
    pannel.add(bouton2);
```

```
    JButton bouton3 = new JButton("Bouton 3");
    pannel.add(bouton3);

    f.getContentPane().add(pannel);
    f.getRootPane().setDefaultButton(bouton3);
    f.setVisible(true);
  }
}
```

Le bouton par défaut est activé par un appui sur la touche Entrée alors que le bouton actif est activé par un appui sur la barre d'espace.

La méthode isDefaultButton() de JButton permet de savoir si le composant est le bouton par défaut.

13.5.3. La classe JToggleButton

Cette classe définit un bouton à deux états : c'est la classe mère des composants JCheckBox et JRadioButton.

La méthode setSelected() héritée de AbstractButton permet de mettre à jour l'état du bouton. La méthode isSelected() permet de connaître cet état.

13.5.4. La classe ButtonGroup

La classe ButtonGroup permet de gérer un ensemble de boutons en garantissant qu'un seul bouton du groupe sera sélectionné.

Pour utiliser la classe ButtonGroup, il suffit d'instancier un objet et d'ajouter des boutons (objet héritant de la classe AbstractButton grâce à la méthode add(). Il est préférable d'utiliser des objets de la classe JToggleButton ou d'une de ces classes filles car elles sont capables de gérer leurs états.

Exemple (code java 1.1) :

```
import javax.swing.*;

public class TestGroupButton1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();

    ButtonGroup groupe = new ButtonGroup();
    JRadioButton bouton1 = new JRadioButton("Bouton 1");
    groupe.add(bouton1);
    pannel.add(bouton1);
    JRadioButton bouton2 = new JRadioButton("Bouton 2");
    groupe.add(bouton2);
    pannel.add(bouton2);
    JRadioButton bouton3 = new JRadioButton("Bouton 3");
    groupe.add(bouton3);
    pannel.add(bouton3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

13.5.5. Les cases à cocher : la classe JCheckBox

Les constructeurs sont les suivants :

Constructeur	Rôle
JCheckBox(String)	précise l'intitulé
JCheckBox(String, boolean)	précise l'intitulé et l'état
JCheckBox(Icon)	précise une icône comme intitulé
JCheckBox(Icon, boolean)	précise une icône comme intitulé et l'état
JCheckBox(String, Icon)	précise un texte et une icône comme intitulé
JCheckBox(String, Icon, boolean)	précise un texte et une icône comme intitulé et l'état

Un groupe de cases à cocher peut être défini avec la classe ButtonGroup. Dans ce cas, un seul composant du groupe peut être sélectionné. Pour l'utiliser, il faut créer un objet de la classe ButtonGroup et utiliser la méthode add() pour ajouter un composant au groupe.

Exemple (code java 1.1) :

```
import javax.swing.*;

public class TestJCheckBox1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();

    JCheckBox bouton1 = new JCheckBox("Bouton 1");
    pannel.add(bouton1);
    JCheckBox bouton2 = new JCheckBox("Bouton 2");
    pannel.add(bouton2);
    JCheckBox bouton3 = new JCheckBox("Bouton 3");
    pannel.add(bouton3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

13.5.6 Les boutons radio : la classe JRadioButton

Les constructeurs sont les même que ceux de la classe JCheckBox.

Exemple (code java 1.1) :

```
import javax.swing.*;

public class TestJRadioButton1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();
    JRadioButton bouton1 = new JRadioButton("Bouton 1");
    pannel.add(bouton1);
```

```
    JRadioButton bouton2 = new JRadioButton("Bouton 2");
    pannel.add(bouton2);
    JRadioButton bouton3 = new JRadioButton("Bouton 3");
    pannel.add(bouton3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

Pour regrouper plusieurs boutons radio, il faut utiliser la classe CheckboxGroup

La suite de ce chapitre est en cours d'écriture

14. Les applets

Une appet est une programme Java qui s'éxécute dans un logiciel de navigation supportant java ou dans l'appletviewer du JDK.

Attention : il est recommandé de tester les applets avec l'appletviewer car les navigateurs peuvent prendre l'applet contenu dans leur cache plutôt que la dernière version compilée.

Le mécanisme d'initialisation d'une applet se fait en deux temps :

1. la machine virtuelle java instancie l'objet Applet en utilisant le constructeur par défaut
2. la machine virtuelle java envoie le message init à l'objet Applet

Ce chapitre contient plusieurs sections :

14.1. L'intégration d'applets dans une page HTML

Dans une page HTML, il faut utiliser le tag APPLET avec la syntaxe suivante :

<APPLET CODE=« Exemple.class » WIDTH=200 HEIGHT=300 > </APPLET>

Le nom de l'applet est indiqué entre guillement à la suite du parametre CODE.

Les paramètres WIDTH et HEIGHT fixent la taille de la fenêtre de l'applet dans la page HTML. L'unité est le pixel. Il est préférable de ne pas dépasser 640 * 480 (VGA standard).

Le tag APPLET peut comporter les attributs facultatifs suivants :

Tag	Role
CODEBASE	permet de spécifier le chemin relatif par rapport au dossier de la page contenant l'applet. Ce paramètre suit le paramètre CODE. Exemple : CODE=nomApplet.class CODEBASE=/nomDossier
HSPACE et VSPACE	permettent de fixer la distance en pixels entre l'applet et le texte

ALT	affiche le texte specifié par le parametre lorsque le navigateur ne supporte pas Java ou que son support est désactivé.

Le tag PARAM permet de passer des paramètres à l'applet. Il doit etre inclus entre les tags APPLET et /APPLET.

<PARAM nomParametre value=« valeurParametre »> </APPLET>

La valeur est toujours passée sous forme de chaine de caractères donc entourée de guillemets.

Exemple : <APPLET code=« Exemple.class » width=200 height=300>

Le texte contenu entre <APPLET> et </APPLET> est afficher si le navigateur ne supporte pas java.

14.2. Les méthodes des applets

Une classe dérivée de la classe java.applet.Applet hérite de méthodes qu'il faut redéfinir en fonction des besoins et doit être déclarée public pour fonctionner.

En général, il n'est pas nécessaire de faire un appel explicite aux méthode init(), start(), stop() et destroy() : le navigateur se charge d'appeler ces méthodes en fonction de l'état de la page HTML contenant l'applet.

14.2.1. La méthode init()

Cette méthode permet l'initialisation de l'applet : elle n'est éxécutée qu'une seule et unique fois après le chargement de l'applet.

14.2.2. La méthode start()

Cette méthode est appelée automatiquement après le chargement et l'initialisation (via la méthode init()) lors du premier affichage de l'applet.

14.2.3. La méthode stop()

Le navigateur appelle automatiquement la méthode lorsque l'on quitte la page HTML. Elle interrompt les traitements de tous les processus en cours.

14.2.4. La méthode destroy()

Elle est appelée après l'arrêt de l'applet ou lors de l'arret de la machine virtuelle. Elle libère les ressources et détruit les threads restants

14.2.5. La méthode update()

Elle est appelée à chaque rafraichissement de l'écran ou appel de la méthode repaint(). Elle efface l'écran et appelle la méthode paint(). Ces actions provoquent souvent des scintillements. Il est préférable de redéfinir cette méthode pour qu'elle n'efface plus l'écran :

Exemple :

```
public void update(Graphics g) { paint (g);}
```

14.2.6. La méthode paint()

Cette méthode permet d'afficher le contenu de l'applet à l'écran. Ce refraîchissement peut être provoqué par le navigateur ou par le système d'exploitation si l'ordre des fenêtres ou leur taille ont été modifiés ou si une fenêtre recouvre l'applet.

Exemple :

```
public void paint(Graphics g)
```

La méthode repaint() force l'utilisation de la méthode paint().

Il existe des méthodes dédié à la gestion de la couleur de fond et de premier plan

La méthode setBackground(Color), héritée de Component, permet de définir la couleur de fond d'une applet. Elle attend en paramètre un objet de la classe Color.

La méthode setForeground(Color) fixe la couleur d'affichage par défaut. Elle s'applique au texte et aux graphiques.

Les couleurs peuvent être spécifiées de 3 manières différentes :

utiliser les noms standard predéfinis	Color.nomDeLaCouleur Les noms predéfinis de la classe Color sont : black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow
utiliser 3 nombres entiers représentant le RGB	(Red,Green,Blue : rouge,vert, bleu) Exemple : `Color macouleur = new Color(150,200,250);` `setBackground (macouleur);` `// ou setBackground(150,200,250);`
utiliser 3 nombre de type float utilisant le systeme HSB	(Hue, Saturation, Brightness : teinte, saturation, luminance). Ce système est moins répandu que le RGB mais il permet notamment de modifier la luminance sans modifier les autres caractéristiques Exemple : `setBackground(0.0,0.5,1.0);` dans ce cas 0.0,0.0,0.0 représente le noir et 1.0,1.0,1.0 représente le blanc.

14.2.7. Les méthodes size() et getSize()

L'origine des coordonnées en Java est le coin supérieur gauche. Elles s'expriment en pixels avec le type int.

La détermination des dimensions d'une applet se fait de la façon suivante :

Exemple (code java 1.0) :

```
Dimension dim = size();
int applargeur = dim.width;
int apphauteur = dim.height;
```

Avec le JDK 1.1, il faut utiliser getSize() à la place de size().

Exemple (code java 1.1) :

```
public void paint(Graphics g) {
   super.paint(g);
   Dimension dim = getSize();
   int applargeur = dim.width;
   int apphauteur = dim.height;
   g.drawString("width = "+applargeur,10,15);
   g.drawString("height = "+apphauteur,10,30);
}
```

14.2.8. Les méthodes getCodeBase() et getDocumentBase()

Ces méthodes renvoient respectivement l'emplacement de l'applet sous forme d'adresse Web ou de dossier et l'emplacement de la page HTML qui contient l'applet.

Exemple :

```
public void paint(Graphics g) {
   super.paint(g);
   g.drawString("CodeBase = "+getCodeBase(),10,15);
   g.drawString("DocumentBase = "+getDocumentBase(),10,30);
}
```

14.2.9. La méthode showStatus()

Affiche une message dans la barre de status de l'applet

Exemple :

```
public void paint(Graphics g) {
   super.paint(g);
   showStatus("message à afficher dans la barre d'état");
}
```

14.2.10. La méthode getAppletInfo()

Permet de fournir des informations concernant l'auteur, la version et le copyright de l'applet

Exemple :

```
static final String appletInfo = " test applet : auteur, 1999 \n\nCommentaires";

public String getAppletInfo() {
```

```
   return appletInfo;
}
```

Pour voir les informations, il faut utiliser l'option info du menu Applet de l'appletviewer.

14.2.11. La méthode getParameterInfo()

Permet de fournir des informations sur les paramètres reconnus par l'applet

Le format du tableau est le suivant :

{ {nom du paramètre, valeurs possibles, description} , ... }

Exemple :

```
static final String[][] parameterInfo =
{ {"texte1", "texte1", " commentaires du texte 1" } ,
  {"texte2", "texte2", " commentaires du texte 2" } };

public String[][] getParameterInfo() {
   return parameterInfo;
}
```

Pour voir les informations, il faut utiliser l'option info du menu Applet de l'appletviewer.

14.2.12. La méthode getGraphics()

Elle retourne la zone graphique d'une applet : utile pour dessiner dans l'applet avec des méthodes qui ne possèdent pas le contexte graphique en paramètres (ex : mouseDown ou mouseDrag).

14.2.13. La méthode getAppletContext()

Cette méthode permet l'accès à des fonctionnalités du navigateur.

14.2.14. La méthode setStub()

Cette méthode permet d'attacher l'applet au navigateur.

14.3. Les interfaces utiles pour les applets

14.3.1. L'interface Runnable

Cette interface fournit le comportement nécessaire à un applet pour devenir un thread.

Les méthodes start() et stop() de l'applet peuvent permettre d'arrêter et de démarrer un thread pour permettre de limiter l'usage des ressources machines lorsque la page contenant l'applet est inactive.

14.3.2. L'interface ActionListener

Cette interface permet à l'applet de répondre aux actions de l'utilisateur avec la souris

La méthode actionPerformed() permet de définir les traitements associés aux événements.

Exemple (code java 1.1) :

```
public void actionPerformed(ActionEvent evt) { ... }
```

Pour plus d'information, voir le chapitre Gestion des événements avec Java 1.1

14.3.3. L'interface MouseListener pour répondre à un clic de souris

Exemple (code java 1.1) :

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

public class AppletMouse extends Applet implements MouseListener {
   int nbClick = 0;

   public void init() {
      super.init();
      addMouseListener(this);
   }

   public void mouseClicked(MouseEvent e) {
      nbClick++;
      repaint();
   }

   public void mouseEntered(MouseEvent e) {
   }

   public void mouseExited(MouseEvent e) {
   }

   public void mousePressed(MouseEvent e) {
   }

   public void mouseReleased(MouseEvent e) {
   }

   public void paint(Graphics g) {
      super.paint(g);
      g.drawString("Nombre de clics : " + nbClick, 10, 10);
   }

}
```

Pour plus d'information, voir le chapitre Gestion des événements avec Java 1.1 et celui sur l'interface MouseListener

14.4. La transmission de paramètres à une applet

La méthode getParameter() retourne les paramètres écrits dans la page HTML. Elle retourne une chaine de caractères de type String.

Exemple :

```
Sting parametre;
parametre = getParameter(« nom-parametre »);
```

Si le paramètre n'est pas renseigné dans la page HTML alors getParameter retourne null

Pour utiliser les valeurs des paramètres, il sera souvent nécessaire de faire une conversion de la chaine de caractères dans le type voulu en utilisant les Wrappers

Exemple :

```
String taille;
int hauteur;
taille = getParameter(« hauteur »);
Integer temp = new Integer(taille)
hauteur = temp.intValue();
```

Exemple :

```
int vitesse;
String paramvitesse = getParameter(« VITESSE »);
if (paramvitesse != null) vitesse = Integer.parseInt(paramVitesse);
// parseInt ne fonctionne pas avec une chaine vide
```

Attention : l'appel à la méthode getParameter() dans le constructeur pas défaut lève une exception de type NullPointerException.

Exemple :

```
public MonApplet() {
   String taille;
   taille = getParameter(" message ");
}
```

14.5. Applet et le multimédia

14.5.1. Insertion d'images.

Java supporte deux standards :

- le format GIF de Compuserve qui est beaucoup utilisé sur internet car il génère des fichiers de petite taille contenant des images d'au plus 256 couleurs.
- et le format JPEG. qui convient mieux aux grandes images et à celles de plus de 256 couleurs car le taux de compression avec perte de qualité peut être précisé.

Pour la manipulation des images, le package nécessaire est java.awt.image.

La méthode getImage() possède deux signatures : getImage(URL url) et getImage (URL url, String name).

On procède en deux étapes : le chargement puis l'affichage. Si les paramètres fournies à getImage ne désignent pas une image, aucune exception n'est levée.

La méthode getImage() ne charge pas de données sur le poste client. Celles ci seront chargées quand l'image sera dessinée pour la première fois.

Exemple :

```
public void paint(Graphics g) {
   super.paint(g);
   Image image=null;
   image=getImage(getDocumentBase( ), "monimage.gif"); //chargement de l'image
   g.drawImage(image, 40, 70, this);
}
```

Le sixième paramètre de la méthode drawImage() est un objet qui implémente l'interface ImageObserver. ImageObserver est une interface déclarée dans le package java.awt.image qui sert à donner des informations sur le fichier image. Souvent, on indique this à la place de cette argument représentant l'applet elle même. La classe ImageObserver détecte le chargement et la fin de l'affichage d'une image. La classe Applet contient le comportement qui se charge de faire ces actions d'ou le fait de mettre this.

Pour obtenir les dimensions de l'image à afficher on peut utiliser les méthodes getWidth() et getHeight() qui retourne un nombre entier en pixels.

Exemple :

```
int largeur = 0;
int hauteur = 0;
largeur = image.getWidth(this);
hauteur = image.getHeight(this);
```

14.5.2. Insertion de sons

Seul le format d'extension .AU de Sun est supporté par java. Pour utiliser un autre format, il faut le convertir.

La méthode play() permet de jouer un son.

Exemple :

```
import java.net.URL;

   ...
   try {
      play(new URL(getDocumentBase(), « monson.au »));
   } catch (java.net.MalformedURLException e) {}
```

La méthode getDocumentBase() détermine et renvoie l'URL de l'applet.

Ce mode d'exécution n'est valable que si le son n'est à reproduire qu'une seule fois, sinon il faut utiliser l'interface AudioClip.

Avec trois méthodes, l'interface AudioClip facilite l'utilisation des sons :

- public abstract void play() // jouer une seule fois le fichier
- public abstract void loop() // relancer le son jusqu'à interruption par la méthode stop ou la fin de l'applet
- public abstract void stop() // fin de la reproduction du clip audio

Exemple :

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

public class AppletMusic extends Applet {
   protected AudioClip aC = null;

   public void init() {
      super.init();
      try {
      AppletContext ac = getAppletContext();
      if (ac != null)
         aC = ac.getAudioClip(new URL(getDocumentBase(), "spacemusic.au"));
      else
         System.out.println(" fichier son introuvable ");
      }
      catch (MalformedURLException e) {}
      aC.loop();
   }
}
```

Pour utiliser plusieurs sons dans une applet, il suffit de déclarer plusieurs variables AudioClip.

L'objet retourné par la méthode getAudioClip() est un objet qui implémente l'interface AudioClip défini dans la machine virtuelle car il est très dépendant du système de la plate forme d'exécution.

14.5.3. Animation d'un logo

Exemple :

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

public class AppletAnimation extends Applet implements Runnable {
   Thread thread;
   protected Image tabImage[];
   protected int index;

   public void init() {
      super.init();
      //chargement du tableau d'image
      index = 0;
      tabImage = new Image[2];
      for (int i = 0; i < tabImage.length; i++) {
         String fichier = new String("monimage" + (i + 1) + ".gif ");
         tabImage[i] = getImage(getDocumentBase(), fichier);
      }
   }

   public void paint(Graphics g) {
      super.paint(g);
      // affichage de l'image
      g.drawImage(tabImage[index], 10, 10, this);
   }

   public void run() {
      //traitements éxécuté par le thread
      while (true) {
         repaint();
         index++;
         if (index >= tabImage.length)
            index = 0;
         try {
            thread.sleep(500);
         }
```

```
         catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }

   public void start() {
      //demarrage du tread
      if (thread == null) {
         thread = new Thread(this);
         thread.start();
      }
   }

   public void stop() {
      // arret du thread
      if (thread != null) {
         thread.stop();
         thread = null;
      }
   }

   public void update(Graphics g) {
      //la redéfinition de la méthode permet d'éviter les scintillements
      paint(g);
   }

}
```

La surcharge de la méthode paint() permet d'éviter le scintillement de l'écran due à l'effacement de l'écran et à son rafraichissement. Dans ce cas, seul le rafraichissement est effectué.

14.6. Applet et application (applet pouvant s'éxécuter comme application)

Il faut rajouter une classe main à l'applet, définir une fenêtre qui recevra l'affichage de l'applet, appeler les méthodes init() et start() et afficher la fenetre.

Exemple (code java 1.1) :

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

public class AppletApplication extends Applet implements WindowListener {

   public static void main(java.lang.String[] args) {
      AppletApplication applet = new AppletApplication();
      Frame frame = new Frame("Applet");
      frame.addWindowListener(applet);
      frame.add("Center", applet);
      frame.setSize(350, 250);
      frame.show();
      applet.init();
      applet.start();
   }

   public void paint(Graphics g) {
      super.paint(g);
      g.drawString("Bonjour", 10, 10);
   }

   public void windowActivated(WindowEvent e) { }

   public void windowClosed(WindowEvent e) { }
```

```
   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }

   public void windowDeactivated(WindowEvent e) { }

   public void windowDeiconified(WindowEvent e) { }

   public void windowIconified(WindowEvent e) { }

   public void windowOpened(WindowEvent e) { }

}
```

14.7. Les droits des applets

Une applet est une application Java hébergée sur une machine distante (un serveur Web) et qui s'éxécute, après chargement, sur la machine client équipée d'un navigateur. Ce navigateur controle les accès de l'applet aux ressources locales et ne les autorisent pas systématiquement : chaque navigateur définit sa propre règle.

Le modèle classique de sécurité pour l 'exécution des applets, recommandé par Sun, distingue deux types d'applets : les applets non dignes de confiance (untrusted) qui non pas accès aux ressources locales et externes, les applets dignes de confiance (trusted) qui ont l'accès. Dans ce modèle, une applet est par défaut untrusted.

La signature d'une applet permet de désigner son auteur et de garantir que le code charger par le client est bien celui demandé au serveur. Cependant, une applet signée n'est pas forcement digne de confiance.

La suite de cette section est en cours d'écriture

Partie 3 : les API avancées

Le JDK fourni un certains nombres d'API intégrés au JDK pour des fonctionnalités avancées.

Plusieurs chapitres dans cette partie détaillent les API pour utiliser les fonctionnalités suivantes :

- les collections : propose une revue des classes fournies par le JDK pour gérer des ensembles d'objets
- les flux : explore les classes utiles à la mise en oeuvre d'un des mécanismes de base pour échanger des données
- la sérialisation : présente un mécanisme qui permet de rendre persistant un objet
- l'intéraction avec le réseau :
- l'accès aux bases de données : indique comment utiliser JDBC pour accéder
- l'appel de méthodes distantes : étudie la mise en oeuvre de la technologie RMI pour permettre l'appel de méthodes distantes.
- l'internationalisation : traite d'une façon pratique de la possibilité d'internationnaliser une application
- les composants java beans : examine comment développer et utiliser des composants réutilisables
- logging : indique comment mettre en oeuvre deux API pour la gestion des logs : Log4J du projet open source jakarta et l'API logging du JDK 1.4

15. Les collections

Les collections sont des objets qui permettent de gérer des ensembles d'objets. Ces ensembles de données peuvent être définis avec plusieurs caractéristiques : la possibilité de gérer des doublons, de gérer un ordre de tri, etc. ...

Chaque objet contenu dans une collection est appelé un élément.

15.1. Présentation du framework collection

Dans la version 1 du J.D.K., il n'existe qu'un nombre restreint de classes pour gérer des ensembles de données :

- Vector
- Stack
- Hashtable
- Bitset

L'interface Enumeration permet de parcourir le contenu de ces objets.

Pour combler le manque d'objets adaptés, la version 2 du J.D.K. apporte un framework complet pour gérer les collections. Cette bibliothèque contient un ensemble de classes et interfaces. Elle fourni également un certain nombre de classes abstraites qui implémentent partiellement certaines interfaces.

Les interfaces à utiliser par des objets qui gèrent des collections sont :

- Collection : interface qui est implementée par la plupart des objets qui gèrent des collections
- Map : interface qui définie des méthodes pour des objets qui gèrent des collections sous la forme clé/valeur
- Set : interface pour des objets qui n'autorisent pas la gestion des doublons dans l'ensemble
- List : interface pour des objets qui autorisent la gestion des doublons et un accès direct à un élément
- SortedSet : interface qui étend l'interface Set et permet d'ordonner l'ensemble
- SortedMap : interface qui étend l'interface Map et permet d'ordonner l'ensemble

Certaines méthodes définies dans ces interfaces sont dites optionnelles : leur définition est donc obligatoire mais si l'opération n'est pas supportées alors la méthode doit lever une exception particulière. Ceci permet de réduire le nombre d'interfaces et de répondre au maximum de cas.

Le framework propose plusieurs objets qui implémentent ces interfaces et qui peuvent être directement utilisés :

- HashSet : HashTable qui implémente l'interface Set
- TreeSet : arbre qui implémente l'interface SortedSet
- ArrayList : tableau dynamique qui implémente l'interface List
- LinkedList : liste doublement chaînée (parcours de la liste dans les deux sens) qui implémente l'interface List
- HashMap : HashTable qui implémente l'interface Map
- TreeMap : arbre qui implémente l'interface SortedMap

Le framework définit aussi des interfaces pour faciliter le parcours des collections et leur tri :

- Iterator : interface pour le parcours des collections
- ListIterator : interface pour le parcours des listes dans les deux sens et modifier les éléments lors de ce parcours
- Comparable : interface pour définir un ordre de tri naturel pour un objet
- Comparator : interface pour définir un ordre de tri quelconque

Deux classes existantes dans les précédentes versions du JDK ont été modifiées pour implémenter certaines interfaces du framework :

- Vector : tableau à taille variable qui implémente maintenant l'interface List
- HashTable : table de hashage qui implémente maintenant l'interface Map

Le framework propose la classe Collections qui contient de nombreuses méthodes statiques pour réaliser certaines opérations sur une collection. Plusieurs méthodes unmodifiableXXX() (ou XXX représente une interface d'une collection) permettent de rendre une collection non modifiable. Plusieurs méthodes synchronizedXXX() permettent d'obtenir une version synchronisée d'une collection pouvant ainsi être manipulée de façon sûre par plusieurs threads. Enfin plusieurs méthodes permettent de réaliser des traitements sur la collection : tri et duplication d'une liste, recherche du plus petit et du plus grand élément, etc. ...

Le framework fourni plusieurs classes abstraites qui proposent une implémentation partielle d'une interface pour faciliter la création d'un collection personnalisée : AbstractCollection, AbstractList, AbstractMap, AbstractSequentialList et AbstractSet.

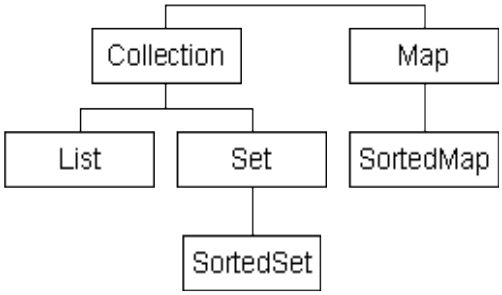
Les objets du framework stockent toujours des références sur les objets contenus dans la collection et non les objets eux mêmes. Ce sont obligatoirement des objets qui doivent être ajoutés dans une collection. Il n'est pas possible de stocker directement des types primitifs : il faut obligatoirement encapsuler ces données dans des wrappers.

Toutes les classes de gestion de collection du framework ne sont pas synchronisées : elles ne prennent pas en charge les traitements multi−threads. Le framework propose des méthodes pour obtenir des objets de gestion de collections qui prennent en charge cette fonctionnalité. Les classes Vector et Hashtable était synchronisée mais l'utilisation d'une collection ne se fait généralement pas de ce contexte. Pour réduire les temps de traitement dans la plupart des cas, elles ne sont pas synchronisées par défaut.

Lors de l'utilisation de ces classes, il est préférable de stocker la référence de ces objets sous la forme d'une interface qu'ils implémentent plutôt que sous leur forme objet. Ceci rend le code plus facile à modifier si le type de l'objet qui gèrent la collection doit être changé.

15.2. Les interfaces des collections

Le framework de java 2 définit 6 interfaces en relation directe avec les collections qui sont regroupées dans deux arborescences :



Le JDK ne fourni pas de classes qui implémentent directement l'interface Collection.

Le tableau ci dessous présente les différentes classes qui implémentent les interfaces de bases Set, List et Map :

	Set	List	Map
Hashtable	HashSet		HashMap

Tableau redimensionnable		ArrayList, Vector	
Arbre	TreeSet		TreeMap
Liste chaînée		LinkedList	
Classes du JDK 1.1		Stack	Hashtable

Pour gérer toutes les situations de façon simple, certaines méthodes peuvent être définies dans une interface comme «optionnelles ». Pour celles ci, les classes qui implémentent une telle interface, ne sont pas obligées d'implémenter du code qui réalise un traitement mais simplement lève une exception si cette fonctionnalité n'est pas supportée.

Le nombre d'interfaces est ainsi grandement réduit.

Cette exception est du type UnsupportedOperationException. Pour éviter de protéger tous les appels de méthodes d'un objet gérant les collections dans un bloc try−catch, cette exception hérite de la classe RuntimeException.

Toutes les classes fournies par le J.D.K. qui implémentent une des interfaces héritant de Collection implémentent toutes les opérations optionnelles.

15.2.1. L'interface Collection

Cette interface définit des méthodes pour des objets qui gèrent des éléments d'une façon assez générale. Elle est la super interface de plusieurs interfaces du framework.

Plusieurs classes qui gèrent une collection implémentent une interface qui hérite de l'interface Collection. Cette interface est une des deux racines de l'arborescence des collections.

Cette interface définit plusieurs méthodes :

Méthode	Rôle
boolean add(Object)	ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
boolean addAll(Collection)	ajoute à la collection tous les éléments de la collection fournie en paramètre
void clear()	supprime tous les éléments de la collection
boolean contains(Object)	indique si la collection contient au moins un élément identique à celui fourni en paramètre
boolean containsAll(Collection)	indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
boolean isEmpty()	indique si la collection est vide
Iterator iterator()	renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection
boolean remove(Object)	supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
boolean removeAll(Collection)	supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
int size()	renvoie le nombre d'éléments contenu dans la collection
Object[] toArray()	renvoie d'un tableau d'objets qui contient tous les éléments de la collection

Cette interface représente un minimum commun pour les objets qui gèrent des collections : ajout d'éléments, suppression d'éléments, vérifier la présence d'un objet dans la collection, parcours de la collection et quelques opérations diverses sur la totalité de la collection.

Ce tronc commun permet entre autre de définir pour chaque objet gérant une collection, un constructeur pour cette objet demandant un objet de type Collection en paramètre. La collection est ainsi initialisée avec les éléments contenus dans la collection fournie en paramètre.

Attention : il ne faut pas ajouter dans une collection une référence à la collection elle−même.

15.2.2. L'interface Iterator

Cette interface définie des méthodes pour des objets capables de parcourir les données d'une collection.

La définition de cette nouvelle interface par rapport à l'interface Enumeration a été justifiée par l'ajout de la fonctionnalité de suppression et la réduction des noms de méthodes.

Méthode	Rôle
boolean hasNext()	indique si il reste au moins à parcourir dans la collection
Object next()	renvoie la prochain élément dans la collection
void remove()	supprime le dernier élément parcouru

La méthode hasNext() est équivalente à la méthode hasMoreElements() de l'interface Enumeration.

La méthode next() est équivalente à la méthode nextElement() de l'interface Enumeration.

La méthode next() lève une exception de type NoSuchElementException si elle est appelée alors que la fin du parcours des éléments est atteinte. Pour éviter la lever de cette exception, il suffit d'appeler la méthode hasNext() et de conditionner avec le résultat l'appel à la méthode next().

Exemple (code java 1.2) :

```
Iterator iterator = collection.Iterator();
while (iterator.hasNext()) {
            System.out.println(“objet = ”+iterator.next());
}
```

La méthode remove() permet de supprimer l'élément renvoyé par le dernier appel à la méthode next(). Il est ainsi impossible d'appeler la méthode remove() sans un appel correspondant à next() : on ne peut pas appeler deux fois de suite la méthode remove().

Exemple (code java 1.2) : suppression du premier élément

```
      Iterator iterator = collection.Iterator();
      if (iterator.hasNext()) {
            iterator.next();
            itérator.remove();
      }
```

Si aucun appel à la méthode next() ne correspond à celui de la méthode remove(), une exception de type IllegalStateException est levée

15.3. Les listes

Une liste est une collection ordonnée d'éléments qui autorise d'avoir des doublons. Etant ordonnée, un élément d'une liste peut être accédé à partir de son index.

15.3.1. L'interface List

Cette interface étend l'interface Collection.

Les collections qui implémentent cette interface autorisent les doublons dans les éléments de la liste. Ils autorisent aussi l'insertion d'éléments null.

L'interface List propose plusieurs méthodes pour un accès à partir d'un index aux éléments de la liste. La gestion de cet index commence à zéro.

Pour les listes, une interface particulière est définie pour assurer le parcours dans les deux sens de la liste et assurer des mises à jour : l'interface ListIterator

Méthode	Rôle
ListIterator listIterator()	renvoie un objet capable de parcourir la liste
Object set (int, Object)	remplace l'élément contenu à la position précisée par l'objet fourni en paramètre
void add(int, Object)	ajouter l'élément fourni en paramètre à la position précisée
Object get(int)	renvoie l'élément à la position précisée
int indexOf(Object)	renvoie l'index du premier élément fourni en paramètre dans la liste ou –1 si l'élément n'est pas dans la liste
ListIterator listIterator()	renvoie un objet pour parcourir la liste et la mettre à jour
List subList(int,int)	renvoie un extrait de la liste contenant les éléments entre les deux index fournis (le premier index est inclus et le second est exclus). Les éléments contenus dans la liste de retour sont des références sur la liste originale. Des mises à jour de ces éléments impactent la liste originale.
int lastIndexOf(Object)	renvoie l'index du dernier élément fourni en paramètre dans la liste ou –1 si l'élément n'est pas dans la liste
Object set(int, Object)	remplace l'élément à la position indiquée avec l'objet fourni

Le framework propose de classes qui implémentent l'interface List : LinkedList et ArrayList.

15.3.2. Les listes chaînées : la classe LinkedList

Cette classe hérite de la classe AbstractSequentialList et implémente donc l'interface List.

Elle représente une liste doublement chaînée.

Cette classe possède un constructeur sans paramètre et un qui demande une collection. Dans ce dernier cas, la liste sera initialisée avec les éléments de la collection fournie en paramètre.

Exemple (code java 1.2) :

```
LinkedList listeChainee = new LinkedList();
Iterator iterator = listeChainee.Iterator();
listeChainee.add(“element 1”);
listeChainee.add(“element 2”);
listeChainee.add(“element 3”);
while (iterator.hasNext()) {
      System.out.println(“objet = ”+iterator.next());
}
```

Une liste chaînée gère une collection de façon ordonnée : l'ajout d'un élément peut se faire à la fin de la collection ou après n'importe quel élément. Dans ce cas, l'ajout est lié à la position courante lors d'un parcours.

Pour répondre à ce besoin, l'interface qui permet le parcours de la collection est une sous classe de l'interface Iterator : l'interface ListIterator.

Comme les iterator sont utilisés pour faire des mises à jour dans la liste, une exception de type CurrentModificationException levé si un iterator parcours la liste alors qu'un autre fait des mises à jour (ajout ou suppression d'un élément dans la liste).

Pour gérer facilement cette situation, il est préférable si l'on sait qu'il y ait des mises à jour à faire de n'avoir qu'un seul iterator qui soit utilisé.

Plusieurs méthodes pour ajouter, supprimer ou obtenir le premier ou le dernier élément de la liste permettent d'utiliser cette classe pour gérer une pile :

Méthode	Rôle
void addFirst(Object)	insère l'objet en début de la liste
void addLast(Object)	insère l'objet en fin de la liste
Object getFirst()	renvoie le premier élément de la liste
Object getLast()	renvoie le dernier élément de la liste
Object removeFirst()	supprime le premier élément de la liste et renvoie le premier élément
Object removeLast()	supprime le dernier élément de la liste et renvoie le premier élément

De par les caractéristiques d'une liste chaînée, il n'existe pas de moyen d'obtenir un élément de la liste directement. Pourtant, la méthode contains() permet de savoir si un élément est contenu dans la liste et la méthode get() permet d'obtenir l'élément à la position fournie en paramètre. Il ne faut toutefois pas oublier que ces méthodes parcours la liste jusqu'à obtention du résultat, ce qui peut être particulièrement gourmand en terme de temps de réponse surtout si la méthode get() est appelée dans une boucle.

Pour cette raison, il ne faut surtout pas utiliser la méthode get() pour parcourir la liste.

La méthode toString() renvoie une chaîne qui contient tous les éléments de la liste.

15.3.3. L'interface ListIterator

Cette interface définie des méthodes pour parcourir la liste dans les deux sens et effectuer des mises à jour qui agissent par rapport à l'élément courant dans le parcours.

En plus des méthodes définies dans l'interface Iterator dont elle hérite, elle définie les méthodes suivantes :

Méthode	Roles
void add(Object)	ajoute un élément dans la liste en tenant de la position dans le parcours
boolean hasPrevious()	indique si il reste au moins un élément à parcourir dans la liste dans son sens inverse
Object previous()	renvoi l'élément précédent dans la liste
void set(Object)	remplace l'élément courante par celui fourni en paramètre

La méthode add() de cette interface ne retourne pas un booléen indiquant que l'ajout à réussi.

Pour ajouter un élément en début de liste, il suffit d'appeler la méthode add() sans avoir appeler une seule fois la méthode next(). Pour ajouter un élément en fin de la liste, il suffit d'appeler la méthode next() autant de fois que nécessaire pour atteindre la fin de la liste et appeler la méthode add(). Plusieurs appels à la méthode add() successifs, ajoute les éléments à la position courante dans l'ordre d'appel de la méthode add().

15.3.4. Les tableaux redimensionnables : la classe ArrayList

Cette classe représente un tableau d'objets dont la taille est dynamique.

Elle hérite de la classe AbstractList donc elle implémente l'interface List.

Le fonctionnement de cette classe est identique à celui de la classe Vector.

La différence avec la classe Vector est que cette dernière est multi thread (toutes ces méthodes sont synchronisées). Pour une utilisation dans un thread unique, la synchronisation des méthodes est inutile et coûteuse. Il est alors préférable d'utiliser un objet de la classe ArrayList.

Elle définit plusieurs méthodes dont les principales sont :

Méthode	Rôle
boolean add(Object)	ajoute un élément à la fin du tableau
boolean addAll(Collection)	ajoute tous les éléments de la collection fournie en paramètre à la fin du tableau
boolean addAll(int, Collection)	ajoute tous les éléments de la collection fournie en paramètre dans la collection à partir de la position précisée
void clear()	supprime tous les éléments du tableau
void ensureCapacity(int)	permet d'augmenter la capacité du tableau pour s'assurer qu'il puisse contenir le nombre d'éléments passé en paramètre
Object get(index)	renvoie l'élément du tableau dont la position est précisée
int indexOf(Object)	renvoie la position de la première occurrence de l'élément fourni en paramètre
boolean isEmpty()	indique si le tableau est vide
int lastIndexOf(Object)	renvoie la position de la dernière occurrence de l'élément fourni en paramètre
Object remove(int)	supprime dans le tableau l'élément fourni en paramètre
void removeRange(int,int)	supprime tous les éléments du tableau de la première position fourni incluse jusqu'à la dernière position fournie exclue

Object set(int, Object)	remplace l'élément à la position indiquée par celui fourni en paramètre
int size()	renvoie le nombre d'élément du tableau
void trimToSize()	ajuste la capacité du tableau sur sa taille actuelle

Chaque objet de type ArrayList gère une capacité qui est le nombre total d'élément qu'il est possible d'insérer avant d'agrandir le tableau. Cette capacité a donc une relation avec le nombre d'élément contenu dans la collection. Lors d'ajout dans la collection, cette capacité et le nombre d'élément de la collection détermine si le tableau doit être agrandi. Si un nombre important d'élément doit être ajouté, il est possible de forcer l'agrandissement de cette capacité avec la méthode ensureCapacity(). Son usage évite une perte de temps liée au recalcul de la taille de la collection. Un constructeur permet de préciser la capacité initiale.

15.4. Les ensembles

Un ensemble (Set) est une collection qui n'autorise pas l'insertion de doublons.

15.4.1. L'interface Set

Cette classe définit les méthodes d'une collection qui n'accepte pas de doublons dans ces éléments. Elle hérite de l'interface Collection mais elle ne définie pas de nouvelle méthode.

Pour déterminer si un élément est déjà inséré dans la collection, la méthode equals() est utilisée.

Le framework propose deux classes qui implémentent l'interface Set : TreeSet et HashSet

Le choix entre ces deux objets est liés à la nécessité de trié les éléments :

- les éléments d'un objet HashSet ne sont pas triés : l'insertion d'un nouvel élément est rapide
- les éléments d'un objet TreeSet sont triés : l'insertion d'un nouvel éléments est plus long

15.4.2. L'interface SortedSet

Cette interface définie une collection de type ensemble triée. Elle hérite de l'interface Set.

Le tri de l'ensemble peut être assuré par deux façons :

- les éléments contenus dans l'ensemble implémentent l'interface Comparable pour définir leur ordre naturel
- il faut fournir au constructeur de l'ensemble un objet Comparator qui définit l'ordre de tri à utiliser

Elle définie plusieurs méthodes pour tirer parti de cette ordre :

Méthode	Rôle
Comparator comparator()	renvoie l'objet qui permet de trier l'ensemble
Object first()	renvoie le premier élément de l'ensemble
SortedSet headSet(Object)	renvoie un sous ensemble contenant tous les éléments inférieurs à celui fourni en paramètre
Object last()	renvoie le dernier élément de l'ensemble

SortedSet subSet(Object, Object)	renvoie un sous ensemble contenant les éléments compris entre le premier paramètre inclus et le second exclus
SortedSet tailSet(Object)	renvoie un sous ensemble contenant tous les éléments supérieurs ou égaux à celui fourni en paramètre

15.4.3. La classe HashSet

Cette classe est un ensemble sans ordre de tri particulier.

Les éléments sont stockés dans une table de hashage : cette table possède une capacité.

Exemple (code java 1.2) :

```
import java.util.*;
public class TestHashSet {
  public static void main(String args[]) {
    Set set = new HashSet();
    set.add("CCCCC");
    set.add("BBBBB");
    set.add("DDDDD");
    set.add("BBBBB");
    set.add("AAAAA");

    Iterator iterator = set.iterator();
    while (iterator.hasNext()) {System.out.println(iterator.next());}
  }
}
```

Résultat :

```
AAAAA
DDDDD
BBBBB
CCCCC
```

15.4.4. La classe TreeSet

Cette classe est un arbre qui représente un ensemble trié d'éléments.

Cette classe permet d'insérer des éléments dans n'importe quel ordre et de restituer ces éléments dans un ordre précis lors de son parcours.

L'implémentation de cette classe insère un nouvel élément dans l'arbre à la position correspondant à celle déterminée par l'ordre de tri. L'insertion d'un nouvel élément dans un objet de la classe TreeSet est donc plus lent mais le tri est directement effectué.

L'ordre utilisé est celui indiqué par les objets insérés si ils implémentent l'interface Comparable pour un ordre de tri naturel ou fournir un objet de type Comparator au constructeur de l'objet TreeSet pour définir l'ordre de tri.

Exemple (code java 1.2) :

```
import java.util.*;

public class TestTreeSet {
  public static void main(String args[]) {
    Set set = new TreeSet();
    set.add("CCCCC");
    set.add("BBBBB");
```

```
    set.add("DDDDD");
    set.add("BBBBB");
    set.add("AAAAA");

    Iterator iterator = set.iterator();
    while (iterator.hasNext()) {System.out.println(iterator.next());}
  }
}
```

Résultat :

```
AAAAA
BBBBB
CCCCC
DDDDD
```

15.5. Les collections gérées sous la forme clé/valeur

Ce type de collection gère les éléments avec deux entités : une clé et une valeur associée. La clé doit être unique donc il ne peut y avoir de doublons. En revanche la même valeur peut être associées à plusieurs clés différentes.

Avant l'apparition du framework collections, la classe dédiée à cette gestion était la classe Hashtable.

15.5.1. L'interface Map

Cette interface est une des deux racines de l'arborescence des collections. Les collections qui implémentent cette interface ne peuvent contenir de doublons. Chaque clé est associée à une valeur et une seule.

Elle définie plusieurs méthodes pour agir sur la collection :

Méthode	Rôle
void clear()	supprime tous les éléments de la collection
boolean containsKey(Object)	indique si la clé est contenue dans la collection
boolean containsValue(Object)	indique si la valeur est contenue dans la collection
Set entrySet()	renvoie un ensemble contenant les valeurs de la collection
Object get(Object)	renvoie la valeur associée à la clé fournie en paramètre
boolean isEmpty()	indique si la collection est vide
Set keySet()	renvoie un ensemble contenant les clés de la collection
Object put(Object, Object)	insère la clé et sa valeur associée fournies en paramètres
void putAll(Map)	insère toutes les clés/valeurs de l'objet fourni en paramètre
Collection values()	renvoie une collection qui contient toutes les éléments des éléments
Object remove(Object)	supprime l'élément dont la clé est fournie en paramètre
int size()	renvoie le nombre d'éléments de la collection

La méthode entrySet() permet d'obtenir un ensemble contenant toutes les clés.

La méthode values() permet d'obtenir une collection contenant toutes les valeurs. La valeur de retour est une Collection et non un ensemble car il peut y avoir des doublons (plusieurs clés peuvent être associées à la même valeur).

Le J.D.K. 1.2 propose deux nouvelles classes qui implémentent cette interface :

- HashMap qui stocke les éléments dans une table de hashage
- TreeMap qui stocke les éléments dans un arbre

La classe HashTable a été mise à jour pour implémenter aussi cette interface.

15.5.2. L'interface SortedMap

Cette interface définie une collection de type Map triée sur la clé. Elle hérite de l'interface Map.

Le tri peut être assuré par deux façons :

- les clés contenues dans la collection implémentent l'interface Comparable pour définir leur ordre naturel
- il faut fournir au constructeur de la collection un objet Comparator qui définit l'ordre de tri à utiliser

Elle définit plusieurs méthodes pour tirer parti de cette ordre :

Méthode	Rôle
Comparator comparator()	renvoie l'objet qui permet de trier la collection
Object first()	renvoie le premier élément de la collection
SortedSet headMap(Object)	renvoie une sous collection contenant tous les éléments inférieurs à celui fourni en paramètre
Object last()	renvoie le dernier élément de la collection
SortedMap subMap(Object, Object)	renvoie une sous collection contenant les éléments compris entre le premier paramètre inclus et le second exclus
SortedMap tailMap(Object)	renvoie une sous collection contenant tous les éléments supérieurs ou égaux à celui fourni en paramètre

15.5.3. La classe Hashtable

Cette section est en cours d'écriture

15.5.4. La classe TreeMap

Cette section est en cours d'écriture

15.5.5. La classe HashMap

Cette section est en cours d'écriture

15.6. Le tri des collections

L'ordre de tri est défini grace à deux interfaces :

- Comparable
- Comparator

15.6.1. L'interface Comparable

Tous les objets qui doivent définir un ordre naturel utilisé par le tri d'une collection avec cet ordre doivent implémenter cette interface.

Cette interface ne définit qu'une seule méthode : int compareTo(Object).

Cette méthode doit renvoyer :

- une valeur entière négative si l'objet courant est inférieur à l'objet fourni
- une valeur entière positive si l'objet courant est supérieur à l'objet fourni
- une valeur nulle si l'objet courant est égal à l'objet fourni

Les classes wrappers, String et Date implémentent cette interface.

15.6.2. L'interface Comparator

Cette interface représente un ordre de tri quelconque. Elle est utile pour permettre le tri d'objet qui n'implémente pas l'interface Comparable ou pour définir un ordre de tri différent de celui défini avec Comparable (l'interface Comparable représente un ordre naturel : il ne peut y en avoir qu'un)

Cette interface ne définit qu'une seule méthode : int compare(Object, Object).

Cette méthode compare les deux objets fournis en paramètre et renvoie :

- une valeur entière négative si le premier objet est inférieur au second
- une valeur entière positive si le premier objet est supérieur au second
- une valeur nulle si les deux objets sont égaux

15.7. Les algorithmes

La classe Collections propose plusieurs méthodes statiques qui effectuer des opérations sur des collections. Ces traitements sont polymorphiques car ils demandent en paramètre un objet qui implémente une interface et retourne une collection.

Méthode	Rôle

void copy(List, List)	copie tous les éléments de la seconde liste dans la première
Enumaration enumeration(Collection)	renvoie un objet Enumeration pour parcourir la collection
Object max(Collection)	renvoie le plus grand élément de la collection selon l'ordre naturel des éléments
Object max(Collection, Comparator)	renvoie le plus grand élément de la collection selon l'ordre naturel précisé par l'objet Comparator
Object min(Collection)	renvoie le plus petit élément de la collection selon l'ordre naturel des éléments
Object min(Collection, Comparator)	renvoie le plus petit élément de la collection selon l'ordre précisé par l'objet Comparator
void reverse(List)	inverse l'ordre de la liste fournie en paramètre
void shuffle(List)	réordonne tous les éléments de la liste de façon aléatoire
void sort(List)	trie la liste dans un ordre ascendant selon l'ordre naturel des éléments
void sort(List, Comparator)	trie la liste dans un ordre ascendant selon l'ordre précisé par l'objet Comparator

Si la méthode sort(List) est utilisée, il faut obligatoirement que les éléments inclus dans la liste implémentent tous l'interface Comparable sinon une exception de type ClassCastException est levée.

Cette classe propose aussi plusieurs méthodes pour obtenir une version multi−thread ou non modifiable des principales interfaces des collections : Collection, List, Map, Set, SortedMap, SortedSet

- XXX synchronizedXXX(XXX) pour obtenir une version multi−thread des objets implémentant l'interface XXX
- XXX unmodifiableXXX(XXX) pour obtenir une version non modifiable des objets implémentant l'interface XXX

Exemple (code java 1.2) :

```
import java.util.*;

public class TestUnmodifiable{
    public static void main(String args[])
    {
      List list = new LinkedList();

      list.add("1");
      list.add("2");
      list = Collections.unmodifiableList(list);

      list.add("3");
    }
}
```

Résultat :

```
C:\>java TestUnmodifiable
Exception in thread "main" java.lang.UnsupportedOperationException
        at java.util.Collections$UnmodifiableCollection.add(Unknown Source)
        at TestUnmodifiable.main(TestUnmodifiable.java:13)
```

L'utilisation d'une méthode synchronizedXXX() renvoie une instance de l'objet qui supporte la synchronisation pour les opérations d'ajout et de suppression d'éléments. Pour le parcours de la collection avec un objet Iterator, il est nécessaire de synchroniser le bloc de code utilisé pour le parcours. Il est important d'inclure aussi dans ce bloc l'appel à la méthode pour obtenir l'objet de type Iterator utilisé pour le parcours.

Exemple (code java 1.2) :

```
import java.util.*;

public class TestSynchronized{
    public static void main(String args[])
    {
      List maList = new LinkedList();

      maList.add("1");
      maList.add("2");
      maList.add("3");
      maList = Collections.synchronizedList(maList);

      synchronized(maList) {
          Iterator i = maList.iterator();
          while (i.hasNext())
               System.out.println(i.next());
          }
     }
}
```

15.8. Les exceptions du framework

L'exception de type UnsupportedOperationException est levée lorsque qu'une opération optionnelle n'est pas supportée par l'objet qui gère la collection.

L'exception ConcurrentModificationException est levée lors du parcours d'une collection avec un objet Iterator et que cette collection subi une modification structurelle.

16. Les flux

Un programme a souvent besoin d'échanger des informations pour recevoir des données d'une source ou pour envoyer des données vers un destinataire.

La source et la destination de ces échanges peuvent être de nature multiple : un fichier, une socket réseau, un autre programme, etc ...

De la même façon, la nature des données échangées peut être diverse : du texte, des images, du son, etc ...

16.1. Présentation des flux

Les flux (stream en anglais) permettent d'encapsuler ces processus d'envoie et de réception de données. Les flux traitent toujours les données de façon séquentielle.

En java, les flux peuvent être divisés en plusieurs catégories :

- les flux d'entrée (input stream) et les flux de sortie (output stream)
- les flux de traitement de caractères et les flux de traitement d'octets

Java définit des flux pour lire ou écrire des données mais aussi des classes qui permettent de faire des traitements sur les données du flux. Ces classes doivent être associées à un flux de lecture ou d'écriture et sont considérées comme des filtres. Par exemple, il existe des filtres qui permettent de mettre les données traitées dans un tampon (buffer) pour les traiter par lots.

Toutes ces classes sont regroupées dans le package java.io.

16.2. Les classes de gestion des flux

Ce qui déroute dans l'utilisation de ces classes, c'est leur nombre et la difficulté de choisir celle qui convient le mieux en fonction des besoins. Pour faciliter ce choix, il faut comprendre la dénomination des classes : cela permet de sélectionner la ou les classes adaptées aux traitements à réaliser.

Le nom des classes se décompose en un préfixe et un suffixe. Il y a quatre suffixes possibles en fonction du type de flux (flux d'octets ou de caractères) et du sens du flux (entrée ou sortie).

	Flux d'octets	Flux de caractères
Flux d'entrée	InputStream	Reader
Flux de sortie	OutputStream	Writer

Il exite donc quatre hiérarchies de classes qui encapsulent des types de flux particuliers. Ces classes peuvent être séparées en deux séries de deux catégories différentes : les classes de lecture et d'écriture et les classes permettant la lecture de

caractères ou d'octets.

- les sous classes de Reader sont des types de flux en lecture sur des ensembles de caractères
- les sous classes de Writer sont des types de flux en écriture sur des ensembles de caractères
- les sous classes de InputStream sont des types de flux en lecture sur des ensembles d'octets
- les sous classes de OutputStream sont des types de flux en écriture sur des ensembles d'octets

Pour le préfixe, il faut distinguer les flux et les filtres. Pour les flux, le préfixe contient la source ou la destination selon le sens du flux.

Préfixe du flux	source ou destination du flux
ByteArray	tableau d'octets en mémoire
CharArray	tableau de caractères en mémoire
File	fichier
Object	objet
Pipe	pipeline entre deux threads
String	chaîne de caractères

Pour les filtres, le préfixe contient le type de traitement qu'il effectue. Les filtres n'existent pas obligatoirement pour des flux en entrée et en sortie.

Type de traitement	Préfixe de la classe	En entrée	En sortie
Mise en tampon	Buffered	Oui	Oui
Concaténation de flux	Sequence	Oui pour flux d'octets	Non
Conversion de données	Data	Oui pour flux d'octets	Oui pour flux d'octets
Numérotation des lignes	LineNumber	Oui pour les flux de caractères	Non
Lecture avec remise dans le flux des données	PushBack	Oui	Non
Impression	Print	Non	Oui
Sérialisation	Object	Oui pour flux d'octets	Oui pour flux d'octets
Conversion octets/caractères	InputStream / OutputStream	Oui pour flux d'octets	Oui pour flux d'octets

- Buffered : ce type de filtre permet de mettre les données du flux dans un tampon. Il peut être utilisé en entrée et en sortie
- Sequence : ce filtre permet de fusionner plusieurs flux.
- Data : ce type de flux permet de traiter les octets sous forme de type de données
- LineNumber : ce filtre permet de numéroter les lignes contenues dans le flux
- PushBack : ce filtre permet de remettre des données lues dans le flux
- Print : ce filtre permet de réaliser des impressions formatées
- Object : ce filtre est utilisé par la sérialisation
- InputStream / OuputStream : ce filtre permet de convertir des octets en caractères

La package java.io définit ainsi plusieurs classes :

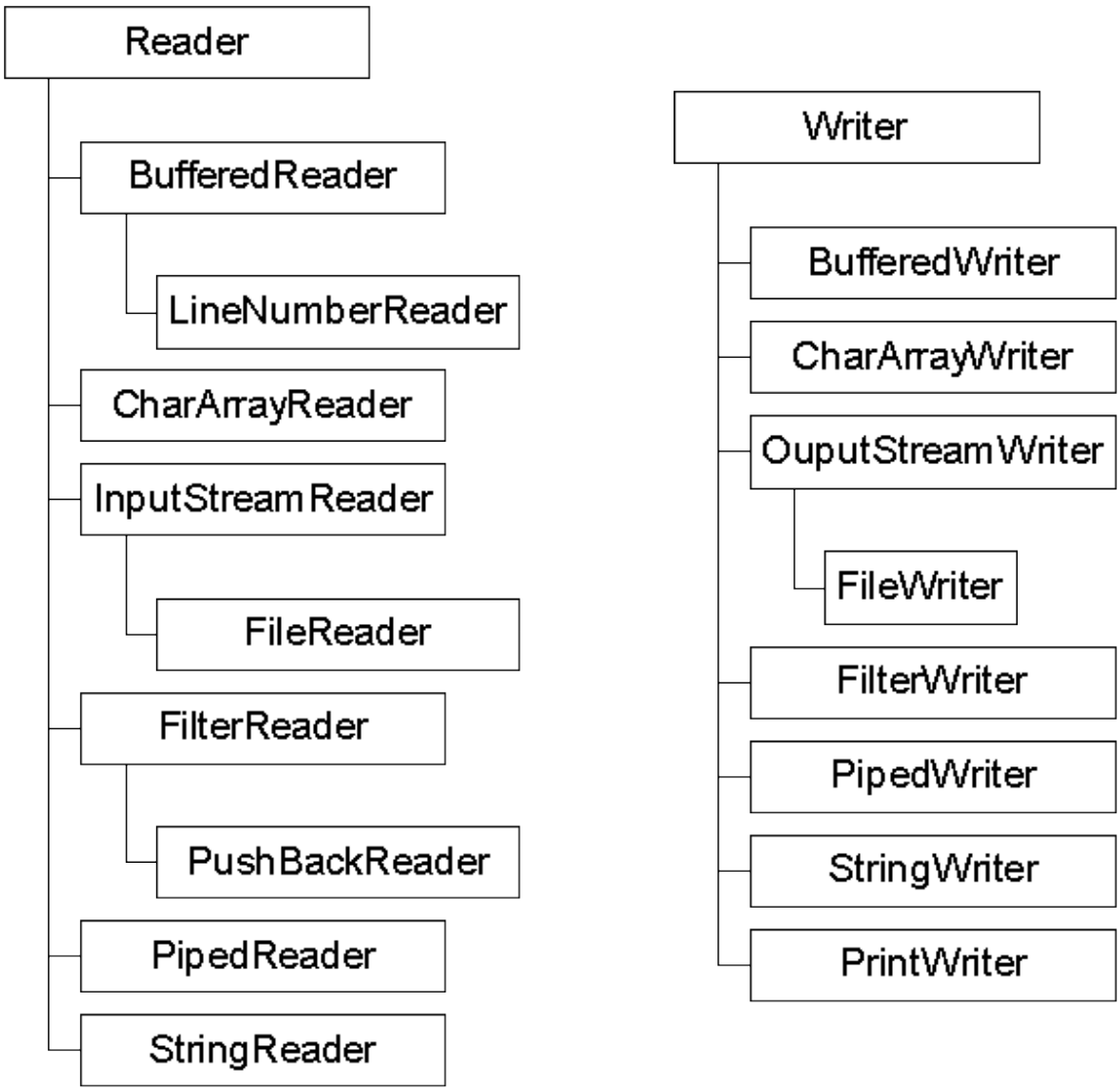
	Flux en lecture	Flux en sortie
Flux de caractères	BufferedReader CharArrayReader FileReader InputStreamReader LineNumberReader PipedReader PushbackReader StringReader	BufferedWriter CharArrayWriter FileWriter OutputStreamWriter PipedWriter StringWriter
Flux d'octets	BufferedInputStream ByteArrayInputStream DataInputStream FileInputStream ObjectInputStream PipedInputStream PushbackInputStream SequenceInputStream	BufferedOutputStream ByteArrayOutputStream DataOuputStream FileOutputStream ObjetOutputStream PipedOutputStream PrintStream

16.3. Les flux de caractères

Ils transportent des données sous forme de caractères : java les gèrent avec le format Unicode qui code les caractères sur 2 octets.

Ce type de flux a été ajouté à partir du JDK 1.1.

Les classes qui gèrent les flux de caractères héritent d'une des deux classes abstraites Reader ou Writer. Il existe de nombreuses sous classes pour traiter les flux de caractères.



16.3.1. La classe Reader

C'est une classe abstaite qui est la classe mère de toutes les classes qui gèrent des flux de caractères en lecture.

Cette classe définit plusieurs méthodes :

Méthodes	Rôles
boolean markSupported()	indique si le flux supporte la possibilité de marquer des positions
boolean ready()	indique si le flux est prêt à être lu
close()	ferme le flux et libère les ressources qui lui étaient associées
int read()	renvoie le caractère lu ou −1 si la fin du flux est atteinte.
int read(char[])	lire plusieurs caractères et les mettre dans un tableau de caractères
int read(char[], int, int)	lire plusieurs caractères. Elle attend en paramètre : un tableau de caractères qui contiendra les caractères lus, l'indice du premier éléments du tableau qui

	recevra le premier caractère et le nombre de caractères à lire. Elle renvoie le nombre de caractères lus ou −1 si aucun caractère n'a été lu. La tableau de caractères contient les caractères lus.
long skip(long)	saute autant de caractères dans le flux que la valeur fournie en paramètre. Elle renvoie le nombre de caractères sautés.
mark()	permet de marquer une position dans le flux
reset()	retourne dans le flux à la dernière position marquée

16.3.2. La classe Writer

C'est une classe abstaite qui est la classe mère de toutes les classes qui gèrent des flux de caractères en écriture.

Cette classe définit plusieurs méthodes :

Méthodes	Rôles
close()	ferme le flux et libère les ressources qui lui étaient associées
write(int)	écrire le caractère en paramètre dans le flux.
write(char[])	écrire le tableau de caractères en paramètre dans le flux.
write(char[], int, int)	écrire plusieurs caractères. Elle attend en paramètres : un tableau de caractères, l'indice du premier caractère dans le tableau à écrire et le nombre de caractères à écrire.
write(String)	écrire la chaîne de caractères en paramètre dans le flux
write(String, int, int)	écrire une portion d'une chaîne de caractères. Elle attend en paramètre : une chaîne de caractères, l'indice du premier caractère dans la chaîne à écrire et le nombre de caractères à écrire.

16.3.3. Les flux de caractères avec un fichier

Les classes FileReader et FileWriter permettent de gérer des flux de caractères avec des fichiers.

16.3.3.1. Les flux de caractères en lecture sur un fichier

Il faut instancier un objet de la classe FileReader. Cette classe hérite de la classe InputStreamReader et possède plusieurs constructeurs qui peuvent tous lever une exception de type FileNotFoundException:

Constructeur	Rôle
FileInputReader(String)	Créer un flux en lecture vers le fichier dont le nom est précisé en paramètre.
FileInputReader(File)	Idem mais le fichier est précisé avec un objet de type File

Exemple (code java 1.1) :
`FileReader fichier = new FileReader(«monfichier.txt»);`

Il existe plusieurs méthodes de la classe FileReader qui permettent de lire un ou plusieurs caractères dans le flux. Toutes ces méthodes sont héritées de la classe Reader et peuvent toutes lever l'exception IOException.

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

16.3.3.2. Les flux de caractères en écriture sur un fichier

Il faut instancier un objet de la classe FileWriter qui hérite de la classe OuputStreamWriter. Cette classe possède plusieurs constructeurs :

Constructeur	Rôle
FileWriter(String)	Si le nom du fichier précisé n'existe pas alors le fichier sera créé. Si il existe et qu'il contient des données celles ci seront écrasées.
FileWriter(File)	Idem mais le fichier est précisé avec un objet de la classe File.
FileWriter(String, boolean)	Le booléen permet de préciser si les données seront ajoutées au fichier (valeur true) ou écraseront les données existantes (valeur false)

Exemple (code java 1.1) :
`FileWriter fichier = new FileWriter («monfichier.dat»);`

Il existe plusieurs méthodes de la classe FileWriter héritées de la classe Writer qui permettent d'écrire un ou plusieurs caractères dans le flux.

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

16.3.4. Les flux de caractères tamponnés avec un fichier.

Pour améliorer les performances des flux sur un fichier, la mise en tampon des données lues ou écrites permet de traiter un ensemble de caractères représentant une ligne plutôt que de traiter les données caractères par caractères. Le nombre d'opérations est ainsi réduit.

Les classes BufferedReader et BufferedWriter permettent de gérer des flux de caractères tamponnés avec des fichiers.

16.3.4.1. Les flux de caractères tamponnés en lecture avec un fichier

Il faut instancier un objet de la classe BufferedReader. Cette classe possède plusieurs constructeurs qui peuvent tous lever une exception de type FileNotFoundException:

Constructeur	Rôle
BufferedReader(Reader)	le paramètre fourni doit correspondre au flux à lire.
BufferedReader(Reader, int)	l'entier en paramètre permet de préciser la taille du buffer. Il doit être positif sinon une exception de type IllegalArgumentException est levée.

Exemple (code java 1.1) :

```
BufferedReader fichier  = new BufferedReader(new FileReader("monfichier.txt"));
```

Il existe plusieurs méthodes de la classe BufferedReader héritées de la classe Reader qui permettent de lire un ou plusieurs caractères dans le flux. Toutes ces méthodes peuvent lever une exception de type IOException. Elle définit une méthode supplémentaire pour la lecture :

Méthode	Rôle
String readLine()	lire une ligne de caractères dans le flux. Une ligne est une suite de caractères qui se termine par un retour chariot '\r' ou un saut de ligne '\n' ou les deux.

La classe BufferedReader possède plusieurs méthodes pour gérer le flux héritées de la classe Reader.

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

Exemple (code java 1.1) :

```
import java.io.*;

public class TestBufferedReader {
  protected String source;

  public TestBufferedReader(String source) {
    this.source = source;
    lecture();
  }

  public static void main(String args[]) {
    new TestBufferedReader("source.txt");
  }

  private void lecture() {
    try {
      String ligne ;
      BufferedReader fichier = new BufferedReader(new FileReader(source));

      while ((ligne = fichier.readLine()) != null) {
          System.out.println(ligne);
      }

      fichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

16.3.4.2. Les flux de caractères tamponnés en écriture avec un fichier

Il faut instancier un objet de la classe BufferedWriter. Cette classe possède plusieurs constructeurs :

Constructeur	Rôle
BufferedWriter(Writer)	le paramètre fourni doit correspondre au flux dans lequel les données sont écrites.
BufferedWriter(Writer, int)	l'entier en paramètre permet de préciser la taille du buffer. Il doit être positif sinon une exception IllegalArgumentException est levée.

Exemple (code java 1.1) :

```
BufferedWriter fichier = new BufferedWriter( new FileWriter(«monfichier.txt»));
```

Il existe plusieurs méthodes de la classe BufferedWriter héritées de la classe Writer qui permettent de lire un ou plusieurs caractères dans le flux.

La classe BufferedWriter possède plusieurs méthodes pour gérer le flux :

Méthode	Rôle
flush()	vide le tampon en écrivant les données dans le flux.
newLine()	écrire un séparateur de ligne dans le flux

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

Exemple (code java 1.1) :

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

public class TestBufferedWriter {
  protected String destination;

  public TestBufferedWriter(String destination) {
    this.destination = destination;
    traitement();
  }

  public static void main(String args[]) {
    new TestBufferedWriter("print.txt");
  }

  private void traitement() {
    try {
      String ligne ;
      int nombre = 123;
      BufferedWriter fichier = new BufferedWriter(new FileWriter(destination));

      fichier.write("bonjour tout le monde");
      fichier.newLine();
      fichier.write("Nous sommes le "+ new Date());
      fichier.write(", le nombre magique est " + nombre);

      fichier.close();
```

```
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

16.3.4.3. La classe PrintWriter

Cette classe permet d'écrire dans un flux des données formatées.

Cette classe possède plusieurs constructeurs :

Constructeur	Rôle
PrintWriter(Writer)	Le paramètre fourni précise le flux. Le tampon est automatiquement vidé.
PrintWriter(Writer, boolean)	Le booléen permet de préciser si le tampon doit être automatiquement vidé
PrintWriter(OutputStream)	Le paramètre fourni précise le flux. Le tampon est automatiquement vidé.
PrintWriter(OutputStream, boolean)	Le booléen permet de préciser si le tampon doit être automatiquement vidé

Exemple (code java 1.1) :

```
PrintWriter fichier = new PrintWriter( new FileWriter(«monfichier.txt»));
```

Il existe de nombreuses méthodes de la classe PrintWriter qui permettent d'écrire un ou plusieurs caractères dans le flux en les formatant. Les méthodes write() sont héritées de la classe Writer. Elle définit plusieurs méthodes pour envoyer des données formatées dans le flux :

- print(...)

  Plusieurs méthodes print acceptent des données de différents types pour les convertir en caractères et les écrire dans le flux

- println()

  Cette méthode permet de terminer la ligne courante dans le flux en y écrivant un saut de ligne.

- println (...)

  Plusieurs méthodes println acceptent des données de différents types pour les convertir en caractères et les écrire dans le flux avec une fin de ligne.

La classe PrintWriter possède plusieurs méthodes pour gérer le flux :

Méthode	Rôle
flush()	Vide le tampon en écrivant les données dans le flux.

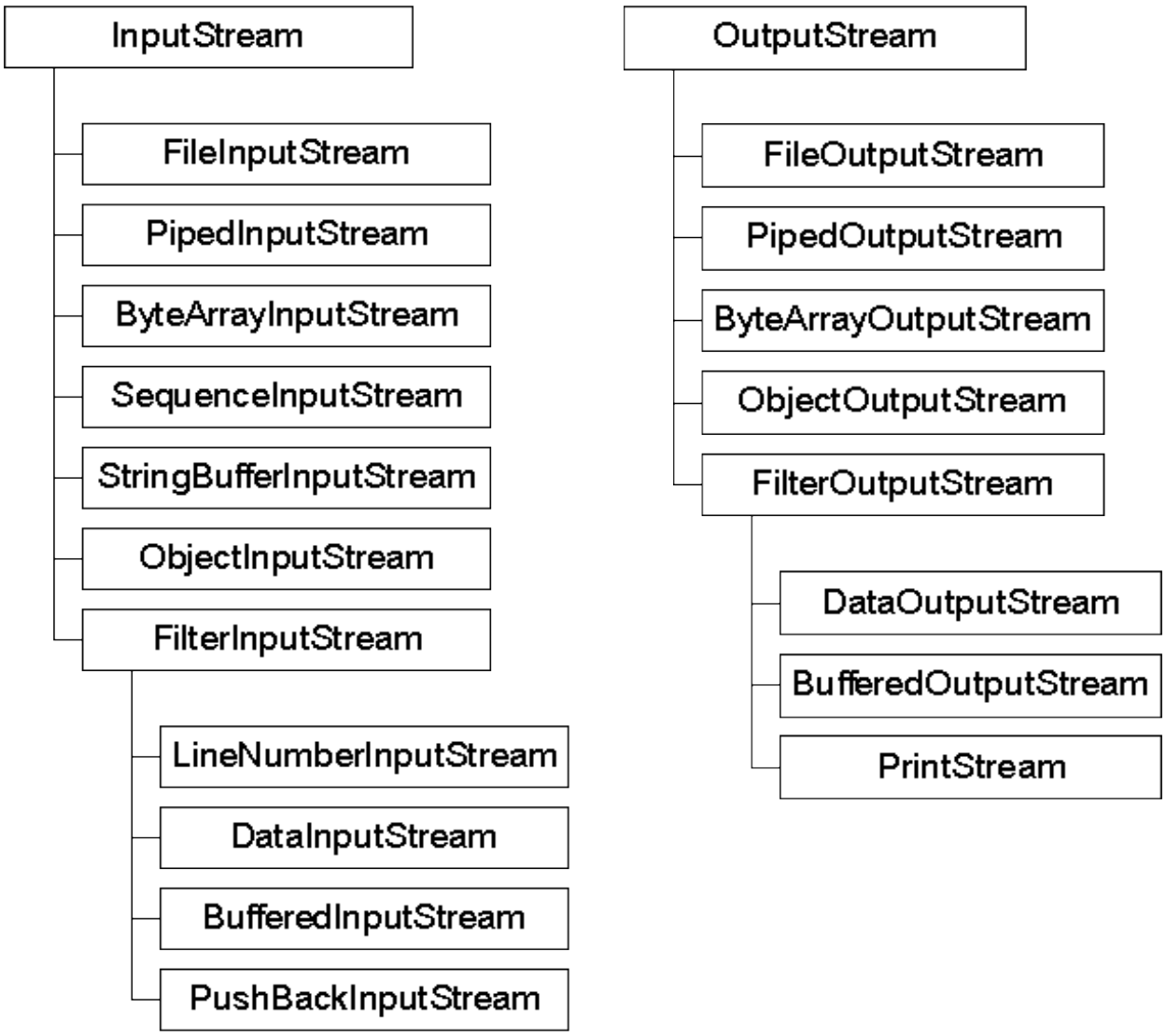
Exemple (code java 1.1) :

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

public class TestPrintWriter {
  protected String destination;

  public TestPrintWriter(String destination) {
    this.destination = destination;
    traitement();
  }

  public static void main(String args[]) {
    new TestPrintWriter("print.txt");
  }

  private void traitement() {
    try {
      String ligne ;
      int nombre = 123;
      PrintWriter fichier = new PrintWriter(new FileWriter(destination));

      fichier.println("bonjour tout le monde");
      fichier.println("Nous sommes le "+ new Date());
      fichier.println("le nombre magique est " + nombre);

      fichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

16.4. Les flux d'octets

Ils transportent des données sous forme d'octets. Les flux de ce type sont capables de traiter toutes les données.

Les classes qui gèrent les flux d'octets héritent d'une des deux classes abstraites InputStream ou OutputStream. Il existe de nombreuses sous classes pour traiter les flux d'octets.



16.4.1. Les flux d'octets avec un fichier.

Les classes FileInputStream et FileOutputStream permettent de gérer des flux d'octets avec des fichiers.

16.4.1.1. Les flux d'octets en lecture sur un fichier

Il faut instancier un objet de la classe FileInputStream. Cette classe possède plusieurs constructeurs qui peuvent tous lever l'exception FileNotFoundException:

Constructeur	Rôle
FileInputStream(String)	Ouvre un flux en lecture sur le fichier dont le nom est donné en paramètre
FileInputStream(File)	Idem mais le fichier est précisé avec un objet de type File

Exemple (code java 1.1) :
`FileInputStream fichier = new FileInputStream(«monfichier.dat»);`

Il existe plusieurs méthodes de la classe FileInputStream qui permettent de lire un ou plusieurs octets dans le flux. Toutes ces méthodes peuvent lever l'exception IOException.

- int read()

  Cette méthode envoie la valeur de l'octet lu ou −1 si la fin du flux est atteinte.

Exemple (code java 1.1) :
`int octet = 0;` `while (octet != 1 ) {` `   octet = fichier.read();` `}`

- int read(byte[], int, int)

  Cette méthode lit plusieurs octets. Elle attend en paramètre : un tableau d'octets qui contiendra les octets lus, l'indice du premier éléments du tableau qui recevra le premier octet et le nombre d'octets à lire.

  Elle renvoie le nombre d'octets lus ou −1 si aucun octet n'a été lus. La tableau d'octets contient les octets lus.

La classe FileInputStream possède plusieurs méthodes pour gérer le flux :

Méthode	Rôle
long skip(long)	saute autant d'octets dans le flux que la valeur fournie en paramètre. Elle renvoie le nombre d'octets sautés.
close()	ferme le flux et libère les ressources qui lui étaient associées
int available()	retourne le nombre d'octets qu'il est encore possible de lire dans le flux

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

16.4.1.2. Les flux d'octets en écriture sur un fichier

Il faut instancier un objet de la classe FileOutputStream. Cette classe possède plusieurs constructeurs :

Constructeur	Rôle
FileOutputStream(String)	Si le fichier précisé n'existe pas, il sera créé. Si il existe et qu'il contient des données celles ci seront écrasées.
FileOutputStream(String, boolean)	Le booléen permet de préciser si les données seront ajoutées au fichier (valeur true) ou écraseront les données existantes (valeur false)

Exemple (code java 1.1) :
`FileOuputStream fichier = new FileOutputStream(«monfichier.dat»);`

Il existe plusieurs méthodes de la classe FileOutputStream qui permettent de lire un ou plusieurs octets dans le flux.

- write(int)

  Cette méthode écrit l'octet en paramètre dans le flux.

- write(byte[])

  Cette méthode écrit plusieurs octets. Elle attend en paramètre : un tableau d'octets qui contient les octets à écrire : tous les éléments du tableau sont écrits.

- write(byte[], int, int)

  Cette méthode écrit plusieurs octets. Elle attend en paramètre : un tableau d'octets qui contient les octets à écrire, l'indice du premier éléments du tableau d'octets à écrire et le nombre d'octets à écrire.

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

Exemple (code java 1.1) :

```
import java.io.*;

public class CopieFichier {
  protected String source;
  protected String destination;

  public CopieFichier(String source, String destination) {
    this.source = source;
    this.destination = destination;
    copie();
  }

  public static void main(String args[]) {
    new CopieFichier("source.txt","copie.txt");
  }

  private void copie() {
    try {
      FileInputStream fis = new FileInputStream(source);
      FileOutputStream fos = new FileOutputStream(destination);
      while(fis.available() > 0) fos.write(fis.read());
      fis.close();
      fos.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

16.4.2. Les flux d'octets tamponnés avec un fichier.

Pour améliorer les performances des flux sur un fichier, la mise en tampon des données lues ou écrites permet de traiter un ensemble d'octets plutôt que de traiter les données octets par octets. Le nombre d'opérations est ainsi réduit.

16.5. La classe File

Les fichiers et les répertoires sont encapsulés dans la classe File du package java.io. Il n'existe pas de classe pour traiter les répertoires car ils sont considérés comme des fichiers. Une instance de la classe File est une représentation logique d'un fichier ou d'un répertoire qui peut ne pas exister physiquement sur le disque.

Si le fichier ou le répertoire existe, de nombreuses méthodes de la classe File permettent d'obtenir des informations sur le fichier. Sinon plusieurs méthodes permettent de créer des fichiers ou des répertoires. Voici une liste des principales méthodes :

Méthode	Rôle
boolean canRead()	indique si le fichier peut être lu
boolean canWrite()	indique si le fichier peut être modifié
boolean createNewFile()	 création d'un nouveau fichier vide
File createTempFile(String, String)	 création d'un nouveau fichier dans le répertoire par défaut des fichiers temporaires. Les deux arguments sont le préfixe et le suffixe du fichier.
File createTempFile(String, String, File)	création d'un nouveau fichier temporaire. Les trois arguments sont le préfixe et le suffixe du fichier et le répertoire.
boolean delete()	détruire le fichier ou le repertoire. Le booléen indique le succés de l'opération
deleteOnExit()	 demande la suppression du fichier à l'arrêt de la JVM
boolean exists()	indique si le fichier existe physiquement
String getAbsolutePath()	renvoie le chemin absolu du fichier
String getPath	renvoie le chemin du fichier
boolean isAbsolute()	indique si le chemin est absolu
boolean isDirectory()	indique si le fichier est un répertoire
boolean isFile()	indique si l'objet représente un fichier
long length()	renvoie la longueur du fichier
String[] list()	renvoie la liste des fichiers et répertoire contenu dans le répertoire
boolean mkdir()	création du répertoire
boolean mkdirs()	création du répertoire avec création des répertoire manquant dans l'arborescence du chemin
boolean renameTo()	renommer le fichier

Depuis la version 1.2 du J.D.K., de nombreuses fonctionnalités ont été ajoutées à cette classe :

- la création de fichiers temporaires (createNewFile, createTempFile, deleteOnExit)
- la gestion des attributs caché et lecture seul (isHidden, isReadOnly)
- des méthodes qui renvoient des objets de type File au lieu de type String (getParentFile, getAbsoluteFile, getCanonicalFile, listFiles)
- une méthode qui renvoie le fichier sous forme d'URL (toURL)

Exemple (code java 1.1) :

```
import java.io.*;

public class TestFile {
  protected String nomFichier;
  protected File fichier;

  public TestFile(String nomFichier) {
    this.nomFichier = nomFichier;
    fichier = new File(nomFichier);
    traitement();
  }

  public static void main(String args[]) {
    new TestFile(args[0]);
  }

  private void traitement() {

    if (!fichier.exists()) {
      System.out.println("le fichier "+nomFichier+"n'existe pas");
      System.exit(1);
    }

    System.out.println(" Nom du fichier    : "+fichier.getName());
    System.out.println(" Chemin du fichier : "+fichier.getPath());
    System.out.println(" Chemin absolu     : "+fichier.getAbsolutePath());
    System.out.println(" Droit de lecture  : "+fichier.canRead());
    System.out.println(" Droite d'ecriture : "+fichier.canWrite());

    if (fichier.isDirectory() ) {
      System.out.println(" contenu du repertoire ");
      String fichiers[] = fichier.list();
      for(int i = 0; i <fichiers.length; i++) System.out.println("  "+fichiers[i]);
    }
  }
}
```

Exemple (code java 1.2) :

```
import java.io.*;

public class TestFile_12 {
  protected String nomFichier;
  protected File fichier;

  public TestFile_12(String nomFichier) {
    this.nomFichier = nomFichier;
    fichier = new File(nomFichier);
    traitement();
  }

  public static void main(String args[]) {
    new TestFile_12(args[0]);
  }

  private void traitement() {

    if (!fichier.exists()) {
      System.out.println("le fichier "+nomFichier+"n'existe pas");
      System.exit(1);
```

```
    }

    System.out.println(" Nom du fichier    : "+fichier.getName());
    System.out.println(" Chemin du fichier : "+fichier.getPath());
    System.out.println(" Chemin absolu     : "+fichier.getAbsolutePath());
    System.out.println(" Droit de lecture  : "+fichier.canRead());
    System.out.println(" Droite d'ecriture : "+fichier.canWrite());

    if (fichier.isDirectory() ) {
      System.out.println(" contenu du repertoire ");
      File fichiers[] = fichier.listFiles();
      for(int i = 0; i <fichiers.length; i++) {

        if (fichiers[i].isDirectory())
          System.out.println("  ["+fichiers[i].getName()+"]");
        else
          System.out.println("  "+fichiers[i].getName());
      }
    }
  }
}
```

16.6. Les fichiers à accès direct

La classe RandomAccesFile permet de gérer les fichiers à accès direct en encapsulant les opérations de lecture/écriture.

Ce chapitre est en cours d'écriture

17. La sérialisation

La sérialisation est un procédé introduit dans le JDK version 1.1 qui permet de rendre un objet persistant. Cet objet est mis sous une forme sous laquelle il pourra être reconstitué à l'identique. Ainsi il pourra être stocké sur un disque dur ou transmis au travers d'un réseau pour le créer dans une autre JVM. C'est le procédé qui est utilisé par RMI. LA serialisation est aussi utilisée par les beans pour sauvegarder leurs états.

Au travers de ce mécanisme, java fourni une façon facile, transparente et standard de réaliser cette opération : ceci permet de facilement mettre en place un mécanisme de persistance. Il est de ce fait inutile de créer un format particulier pour sauvegarder et relire un objet. Le format utilisé est indépendant du système d'exploitation. Ainsi, un objet sérialisé sur un système peut être réutilisé par un autre système pour récréer l'objet.

L'ajout d'un attribut à l'objet est automatiquement pris en compte lors de la sérialisation. Attention toutefois, la déserialisation de l'objet doit se faire avec la classe qui à été utilisée pour la sérialisation.

La sérialisation peut s'appliquer facilement à tous les objets.

17.1. Les classes et les interfaces de la sérialisation

La sérialisation définie l'interface Serializable et les classes ObjectOutputStream et ObjectInputStream

17.1.1. L'interface Serializable

Cette interface ne définie aucune méthode mais permet simplement de marquer une classe comme pouvant être sérialisée.

Tout objet qui doit être sérialiser doit implémenter cette interface ou une de ses classes mères doit l'implémenter.

Si l'on tente de sérialiser un objet qui n'implémente pas l'interface Serializable, une exception NotSerializableException est levée.

Exemple (code java 1.1) : une classe serializable possédant trois attributs

```
public class Personne implements java.io.Serializable {
  private String nom = "";
  private String prenom = "";
  private int taille = 0;

  public Personne(String nom, String prenom, int taille) {
    this.nom = nom;
    this.taille = taille;
    this.prenom = prenom;
  }

  public String getNom() {
    return nom;
  }
```

```
  public void setNom(String nom) {
    this.nom = nom;
  }

  public int getTaille() {
    return taille;
  }

  public void setTaille(int taille) {
    this.taille = taille;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }
}
```

17.1.2. La classe ObjectOuputStream

Cette classe permet de sérialiser un objet.

Exemple (code java 1.1) : sérialisation d'un objet et enregistrement sur le disque dur

```
import java.io.*;

public class SerializerPersonne {

  public static void main(String argv[]) {
    Personne personne = new Personne("Dupond","Jean",175);
    try {
      FileOutputStream fichier = new FileOutputStream("personne.ser");
      ObjectOutputStream oos = new ObjectOutputStream(fichier);
      oos.writeObject(personne);
      oos.flush();
      oos.close();
    }
    catch (java.io.IOException e) {
      e.printStackTrace();
    }
  }
}
```

On définit un fichier avec la classe FileOutputStream. On instancie un objet de classe ObjectOutputStream en lui fournissant en paramètre le fichier : ainsi, le résultat de la sérialisation sera envoyé dans le fichier.

On appel la méthode writeObject en lui passant en paramètre l'objet à sérialiser. On appelle la méthode flush() pour vider le tampon dans le fichier et la méthode close() pour terminer l'opération.

Lors de ces opérations une exception de type IOException peut être levée si un problème intervient avec le fichier.

Après l'exécution de cet exemple, un fichier nommé « personne.ser » est créé. On peut visualiser sont contenu mais surtout pas le modifier car sinon il serait corrompu. En effet, les données contenue dans ce fichier ne sont pas toutes au format caractères.

La classe ObjectOutputStream contient aussi plusieurs méthodes qui permettent de sérialiser des types élémentaires et non des objets : writeInt, writeDouble, writeFloat …

Il est possible dans un même flux d'écrire plusieurs objets les uns à la suite des autres. Ainsi plusieurs objets peuvent être sauvegardés. Dans ce cas, il faut faire attention de relire les objets dans leur ordre d'écriture.

17.1.3. La classe ObjectInputStream

Cette classe permet de déssérialiser un objet.

Exemple (code java 1.1) :

```
import java.io.*;

public class DeSerializerPersonne {

  public static void main(String argv[]) {
    try {
      FileInputStream fichier = new FileInputStream("personne.ser");
      ObjectInputStream ois = new ObjectInputStream(fichier);
      Personne personne = (Personne) ois.readObject();
      System.out.println("Personne : ");
      System.out.println("nom : "+personne.getNom());
      System.out.println("prenom : "+personne.getPrenom());
      System.out.println("taille : "+personne.getTaille());
    }
    catch (java.io.IOException e) {
      e.printStackTrace();
    }
    catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
   }
}
```

Résultat :

```
C:\dej>java DeSerializerPersonne
Personne :
nom : Dupond
prenom : Jean
taille : 175
```

On créer un objet de la classe FileInputStream qui représente le fichier contenant l'objet sérialisé. On créer un objet de type ObjectInputStream en lui passant le fichier en paramètre. Un appèle à la méthode readObject() retourne l'objet avec un type Object. Un cast est nécessaire pour obtenir le type de l'objet. La méthode close() permet de terminer l'opération.

Si la classe a changée entre le moment ou elle a été sérialisée et le moment ou elle est déserialisée, une exception est levée :

Exemple : la classe Personne est modifiée et recompilée

```
C:\temp>java DeSerializerPersonne
java.io.InvalidClassException: Personne; Local class not compatible: stream class
desc serialVersionUID=-2739669178469387642 local class serialVersionUID=39870587
36962107851

atjava.io.ObjectStreamClass.validateLocalClass(ObjectStreamClass.java:4
38)
at java.io.ObjectStreamClass.setClass(ObjectStreamClass.java:482)
at java.io.ObjectInputStream.inputClassDescriptor(ObjectInputStream.java
:785)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:353)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:232)
at java.io.ObjectInputStream.inputObject(ObjectInputStream.java:978)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:369)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:232)
at DeSerializerPersonne.main(DeSerializerPersonne.java:9)
```

Une exception de type StreamCorruptedException peut être levée si le fichier a été corrompu par exemple en le modifiant avec un éditeur.

Exemple : les 2 premiers octets du fichier personne.ser ont été modifiés avec un éditeur hexa

```
C:\temp>java DeSerializerPersonne

java.io.StreamCorruptedException: InputStream does not containa serialized object
at java.io.ObjectInputStream.readStreamHeader(ObjectInputStream.java:731)
at java.io.ObjectInputStream.<init>(ObjectInputStream.java:165)
at DeSerializerPersonne.main(DeSerializerPersonne.java:8)
```

Une exception de type ClassNotFoundException peut être levée si l'objet est transtypé vers une classe qui n'existe plus ou pas au moment de l'exécution.

Exemple (code java 1.1) :

```
C:\temp>rename Personne.class Personne2.class
C:\temp>java DeSerializerPersonne

java.lang.ClassNotFoundException: Personne
at java.io.ObjectInputStream.inputObject(ObjectInputStream.java:981)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:369)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:232)
at DeSerializerPersonne.main(DeSerializerPersonne.java:9)
```

La classe ObjectInputStream possède de la même façon que la classe ObjectOutputStream des méthodes pour lire des données de type primitives : readInt(), readDouble(), readFloat …

Lors de la deserialisation, le constructeur de l'objet n'est jamais utilisé.

17.2 Le mot clé transient

Le contenu des attributs sont visibles dans le flux dans lequel est sérialisé l'objet. Il est ainsi possible pour toute personne ayant accès au flux de voir le contenu de chaque attribut même si ceux si sont private. Ceci peut poser des problèmes de sécurité surtout si les données sont sensibles.

Java introduit le mot clé transient qui précise que l'attribut qu'il qualifie ne doit pas être inclus dans un processus de serailisation et donc de deserialisation.

Exemple (code java 1.1) :

```
...
private transient String codeSecret;
...
```

Lors de la deserialisation, les champs transient sont initialisé avec la valeur null. Ceci peut poser des problèmes à l'objet qui doit gérer cette état pour éviter d'avoir des exceptions de type NullPointerException.

17.3. La sérialisation personnalisée

Il est possible de personnaliser la serialisation d'un objet. Dans ce cas, la classe doit implémenter l'interface Externalizable qui hérite de l'interface Serializable.

17.3.1 L'interface Externalizable

Cette interface définie deux méthode : readExternal() et writeExternal().

Par défaut, la serialisation d'un objet qui implémente cette interface ne prend en compte aucun attribut de l'objet.

Remarque : le mot clé transient est donc inutile avec un classe qui implémente l'interface Externalisable

Cette section est en cours d'écriture

18. L'interaction avec le réseau

Ce chapitre est en cours d'écriture

Java a dès le départ été prévu pour être utilisé sur le web et donc intéragir avec les réseaux. Le package java.net contient un ensemble de classe qui permettent d'utiliser le réseau dans des applications java. Toute les intéractions sur le réseaux utilisent une socket pour réaliser des échanges. Ces échanges peuvent ce faire entre applications sur une même machine ou sur deux machines connectées à un même réseau.

18.1. La classe Socket

La classe Socket représente une socket. Une socket permet d'échanger un ou plusieurs ensembles d'octets regroupés en paquet.

Comme il est possible d'ouvrir plusieurs sockets sur une même machine, une socket est caractérisée par un numéro de port qui est un numéro unique sur la machine. Certains ports sont réservés à des protocoles particuliers : les 1024 premiers ports sont ainsi réservés. Les autres sont librement utilisables dans la mesure ou ils sont associés à une seule socket.

Les sockets fonctionnent par paire sur le principe du client/serveur.

19. L'accès aux bases de données : JDBC

JDBC est l'acronyme de Java DataBase Connectivity et désigne une API définie par Sun pour permettre un accès aux bases de données avec Java.

Ce chapitre présente dans plusieurs sections l'utilisation de cette API :

19.1. Les outils nécessaires pour utiliser JDBC

Les classes de JDBC version 1.0 sont regroupées dans le package java.sql et sont incluses dans le JDK à partir de sa version 1.1. La version 2.0 de cette API est incluse dans la version 1.2 du JDK.

Pour pouvoir utiliser JDBC, il faut un pilote qui est spécifique à la base à laquelle on veut accéder. Avec le JDK, Sun fournit un pilote qui permet l'accès aux bases de données via ODBC.

Ce pilote permet de réaliser l'indépendance de JDBC vis à vis des bases de données.

Pour utiliser le pont JDBC−ODBC sous Window 9x, il faut utiliser ODBC en version 32 bits.

19.2. Les types de pilotes JDBC

Il existe quatre types de pilote JDBC :

1. Type 1 (JDBC−ODBC bridge) : le pont JDBC−ODBC qui s'utilise avec ODBC et un pilote ODBC spécifique pour la base à accéder. Cette solution fonctionne très bien sous Windows. C'est la solution idéale pour des développements avec exécution sous Windows d'une application locale. Cette solution « simple » pour le développement possède plusieurs inconvenients :

   - la multiplication du nombre de couche rend complexe l'architecture (bien que transparent pour le développeur) et détériore un peu les performances
   - lors du deploiement, ODBC et son pilote doivent etre installé sur tous les postes ou l'application va fonctionner.
   - la partie native (ODBC et son pilote) rend l'application moins portable et dépendant d'une plateforme.

2. Type 2 : un driver écrit en java qui appelle l'API native de la base de données

   Ce type de driver convertit les ordres JDBC pour appeler directement les API de la base de données via un pilote natif sur le client. Ce type de driver nécessite aussi l'utilisation de code natif sur le client.

3. Type 3 : un driver écrit en Java utilisant le protocole natif de la base de données

   Ce type de driver utilise un protocole réseau propriétaire spécifique à un base de données. Un serveur dédié reçoit les messages par ce protocole et dialogue directement avec la base de données. Ce type de driver peut être facilement utilisé par une applet mais dans ce cas le serveur intermédiaire doit obligatoirement etre installé sur la machine contenant le serveur web.

4. Type 4 : un driver Java natif

   Ce type de driver, écrit en java, appelle directement le SGBD par le réseau. Ils sont fournis par l'éditeur de la base de données.

Les drivers se présentent souvent sous forme de fichiers jar dont le chemin doit être ajouté au classpath pour permettre au programme de l'utiliser.

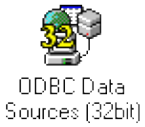
19.3. Enregistrer une base de données dans ODBC sous Windows 9x

Attention : ODBC n'est pas fourni en standard avec Windows 9x.

Pour utiliser un pilote de type 1 (pont ODBC−JDBC) sous Windows 9x, il est nécessaire d'enregistrer la base de données dans ODBC avant de pouvoir l'utiliser.

Pour enregistrer une nouvelle base de données, il faut utiliser l'administrateur de source de données ODBC.

Pour lancer cet application, il faut doubler cliquer sur l'icône "ODBC 32bits" dans le panneau de configuration.



L'onglet "Pilote ODBC" liste l'ensemble des pilotes qui sont installés sur la machine.

Cette section est en cours d'écriture

19.4. Présentation des classes de l'API JDBC

Toutes les classes de JDBC sont dans le package java.sql. Il faut donc l'importer dans tous les programmes devant utiliser JDBC.

Exemple (code java 1.1) :
`import java.sql.*;`

Il y a 4 classes importantes : DriverManager, Connection, Statement (et PreparedStatement), et ResultSet, chacunes correspondant à une étape de l'accès aux données :

Classe	Role
DriverManager	charge et configure le driver de la base de données.
Connection	réalise la connection et l'authentification à la base de données.
Statement (et PreparedStatement)	contient la requète SQL et la transmet à la base de données.
ResultSet	permet de parcourir les informations retournées par la base de données dans le cas d'une sélection de données

Chacunes de ces classes dépend de l'instanciation d'un objet de la précédente classe.

19.5. La connection à une base de données

19.5.1. Le chargement du pilote

Pour se connecter à une base de données via ODBC, il faut tout d'abord charger le pilote JDBC−ODBC qui fait le lien entre les deux.

Exemple (code java 1.1) :

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Pour se connecter à une base en utilisant un driver spécifique, la documentation du driver fournit le nom de la classe à utiliser. Par exemple, si le nom de la classe est jdbc.DriverXXX, le chargement du driver se fera avec le code suivant :

Class.forName("jdbc.DriverXXX");

Exemple (code java 1.1) : Chargement du pilote pour un base PostgreSQL sous Linux

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

Il n'est pas nécessaire de créer une instance de cette classe et de l'enregistrer avec le DriverManager car l'appel à Class.forName le fait automatiquement : ce traitement charge le pilote et créer une instance de cette classe.

La méthode static forName() de la classe Class peut lever l'exception java.lang.ClassNotFoundException.

19.5.2. L'établissement de la connection

Pour se connecter à une base de données, il faut instancier un objet de la classe Connection en lui précisant sous forme d'URL la base à accéder.

Exemple (code java 1.1) : Etablir une connexion sur la base testDB via ODBC

```
String DBurl = "jdbc:odbc:testDB";

con = DriverManager.getConnection(DBurl);
```

La syntaxe URL peut varier d'un type de base de données à l'autres mais elle est toujours de la forme : protocole:sous_protocole:nom

« jbdc » désigne le protocole est vaut toujours « jdbc ». « odbc » désigne le sous protocole qui définit le mécanisme de connection pour un type de bases de données.

Le nom de la base de données doit être celui saisie dans le nom de la source sous ODBC.

La méthode getConnection() peut lever une exception de la classe java.sql.SQLException.

Le code suivant décrit la création d'une connection avec un user et un mot de passe :

Exemple (code java 1.1) :

```
Connection con = DriverManager.getConnection(url, "myLogin", "myPassword");
```

A la place de " myLogin " ; il faut mettre le nom du user qui se connecte à la base et mettre son mot de passe à la place de "myPassword "

Exemple (code java 1.1) :

```
String url = "jdbc:odbc:factures";

Connection con = DriverManager.getConnection(url, "toto", "passwd");
```

La documentation d'un autre driver indiquera le sous protocole à utiliser (le protocole à mettre derrière jdbc dans l'URL).

Exemple (code java 1.1) : Connection à la base PostgreSQL nommée test avec le user jumbo et le mot de passe 12345 sur la machine locale

```
Connection con=DriverManager.getConnection("jdbc:postgresql://localhost/test","jumbo","12345");
```

19.6. Accéder à la base de données

Une fois la connection établie, il est possible d'éxécuter des ordres SQL. Les objets qui peuvent être utilisés pour obtenir des informations sur la base de données sont :

Classe	Role
DatabaseMetaData	informations à propos de la base de données : nom des tables, index, version ...
ResultSet	résultat d'une requète et information sur une table. L'accès se fait enregistrement par enregistrement.
ResultSetMetaData	informations sur les colonnes (nom et type) d'un ResultSet

19.6.1. L'execution de requètes SQL

Les requètes d'interrogation SQL sont éxécutées avec les méthodes d'un objet Statement que l'on obtient à partir d'un objet Connection

Exemple (code java 1.1) :

```
ResultSet résultats = null;
String requète = "SELECT * FROM client";

try {
   Statement stmt = con.createStatement();
   résultats = stmt.éxécuteQuery(requète);
} catch (SQLException e) {
   //traitement de l'exception
}
```

Un objet de la classe Statement permet d'envoyer des requetes SQL à la base. Le création d'un objet Statement s'effectue à partir d'une instance de la classe Connection :

Exemple (code java 1.1) :

```
Statement stmt = con.createStatement();
```

Pour une requete de type interrogation (SELECT), la méthode à utiliser de la classe Statement est éxécuteQuery. Pour des traitements de mise à jour, il faut utiliser la méthode éxécuteUpdate. Lors de l'appel à la méthode d'exécution, il est nécessaire de lui fournir en paramètre la requete SQL sous forme de chaine.

Le résultat d'une requète d'intérrogation est renvoyé dans un objet de la classe ResultSet par la méthode executeQuery().

Exemple (code java 1.1) :

```
ResultSet rs = stmt.éxécuteQuery("SELECT * FROM employe");
```

La méthode executeUpdate() retourne le nombre d'enregistrement qui ont été mis à jour

Exemple (code java 1.1) :

```
...

//insertion d'un enregistrement dans la table client

requète = "INSERT INTO client VALUES (3,'client 3','prenom 3')";
try {
   Statement stmt = con.createStatement();
   int nbMaj = stmt.éxécuteUpdate(requète);
   affiche("nb mise a jour = "+nbMaj);
} catch (SQLException e) {
   e.printStackTrace();
}

...
```

Lorsque la méthode executeUpdate() est utilisée pour éxécuter un traitement de type DDL (Data Defiition Langage : définition de données) comme la création d'un table, elle retourne 0. Si la méthode retourne 0, cela peut signifier deux choses : le traitement de mise à jour n'a affecté aucun enregistrement ou le traitement concernait un traitement de type DDL.

Si l'on utilise éxécuteQuery pour éxécuter une requète SQL ne contenant pas d'ordre SELECT, alors une exception de type SQLException est levée.

Exemple (code java 1.1) :

```
...

requète = "INSERT INTO client VALUES (4,'client 4','prenom 4')";
try {
   Statement stmt = con.createStatement();
   ResultSet résultats = stmt.éxécuteQuery(requète);
} catch (SQLException e) {
   e.printStackTrace();
}

...
```

résultat :

```
java.sql.SQLException: No ResultSet was produced
java.lang.Throwable(java.lang.String)
java.lang.Exception(java.lang.String)
java.sql.SQLException(java.lang.String)
java.sql.ResultSet sun.jdbc.odbc.JdbcOdbcStatement.éxécuteQuery(java.lang.String)
void testjdbc.TestJDBC1.main(java.lang.String [])
```

Attention : dans ce cas la requète est quand même effectuée. Dans l'exemple, un nouvel enregistrement est créé dans la table.

Il n'est pas nécessaire de définir un objet Statement pour chaque ordre SQL : il est possible d'un définir un et de le réutiliser

19.6.2. La classe ResultSet

C'est une classe qui représente une abstraction d'une table qui se compose de plusieurs enregistrements constitués de colonnes qui contiennent les données.

Les principales méthodes pour obtenir des données sont :

Méthode	Role
getInt(int)	retourne le contenu de la colonne dont le numéro est passé en paramètre sous forme d'entier.
getInt(String)	retourne le contenu de la colonne dont le nom est passé en paramètre sous forme d'entier.
getFloat(int)	retourne le contenu de la colonne dont le numéro est passé en paramètre sous forme de nombre flottant.
getFloat(String)	
getDate(int)	retourne le contenu de la colonne dont le numéro est passé en paramètre sous forme de date.
getDate(String)	
next()	se déplace sur le prochain enregistrement : retourne false si la fin est atteinte
Close()	ferme le ResultSet
getMetaData()	retourne un objet ResultSetMetaData associé au ResultSet.

La méthode getMetaData() retourne un objet de la classe ResultSetMetaData qui permet d'obtenir des informations sur le résultat de la requète. Ainsi, le nombre de colonne peut être obtenu grace à la méthode getColumnCount de cet objet.

Exemple (code java 1.1) :

```
ResultSetMetaData rsmd;
rsmd = results.getMetaData();
nbCols = rsmd.getColumnCount();
```

La méthode next() déplace le curseur sur le prochain enregistrement. Le curseur pointe initialement juste avant le premier enregistrement : il est nécessaire de faire un premier appel à la méthode next() pour ce placer sur le premier enregistrement.

Des appels successifs à next permettent de parcourir l'ensemble des enregistrements.

Elle retourne false lorsqu'il n'y a plus d'enregistrement. Il faut toujours protéger le parcours d'une table dans un bloc de capture d'exception

Exemple (code java 1.1) :

```
//parcours des données retournées

try {
   ResultSetMetaData rsmd = résultats.getMetaData();
   int nbCols = rsmd.getColumnCount();
   boolean encore = résultats.next();
   while (encore) {
      for (int i = 1; i <= nbCols; i++)
```

```
         System.out.print(résultats.getString(i) + " ");
      System.out.println();
      encore = résultats.next();
   }
   résultats.close();
} catch (SQLException e) {
   //traitement de l'exception
}
```

Les méthodes getXXX() permettent d'extraire les données selon leur type spécifiée par XXX tel que getString(), getDouble(), getInteger(), etc … . Il existe deux formes de ces méthodes : indiquer le numéro la colonne en paramètre (en commençant par 1) ou indiquer le nom de la colonne en paramètre. La première méthode est plus efficace mais peut générer plus d'erreurs à l'éxecution notamment si la structure de la table évolue.

Attention : il est important de noter que ce numéro de colonne fourni en paramètre fait référence au numéro de colonne de l'objet resultSet (celui correspondant dans l'ordre SELECT)et non au numéro de colonne de la table.

La méthode getString() permet d'obtenir la valeur d'un champ de n'importe quel type.

19.6.3. Exemple complet de mise à jour et de sélection sur une table

Exemple (code java 1.1) :

```
import java.sql.*;

public class TestJDBC1 {

   private static void affiche(String message) {
      System.out.println(message);
   }

   private static void arret(String message) {
      System.err.println(message);
      System.exit(99);
   }

   public static void main(java.lang.String[] args) {
      Connection con = null;
      ResultSet résultats = null;
      String requète = "";

      // chargement du pilote
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      } catch (ClassNotFoundException e) {
         arret("Impossible de charger le pilote jdbc:odbc");
      }

      //connection a la base de données

      affiche("connection a la base de données");
      try {

         String DBurl = "jdbc:odbc:testDB";
         con = DriverManager.getConnection(DBurl);
      } catch (SQLException e) {
         arret("Connection à la base de données impossible");
      }

      //insertion d'un enregistrement dans la table client
      affiche("creation enregistrement");

      requète = "INSERT INTO client VALUES (3,'client 3','client 4')";
```

```
      try {
         Statement stmt = con.createStatement();
         int nbMaj = stmt.éxécuteUpdate(requète);
         affiche("nb mise a jour = "+nbMaj);
      } catch (SQLException e) {
          e.printStackTrace();
      }

      //creation et execution de la requète
      affiche("creation et execution de la requète");
      requète = "SELECT * FROM client";

      try {
         Statement stmt = con.createStatement();
         résultats = stmt.éxécuteQuery(requète);
      } catch (SQLException e) {
         arret("Anomalie lors de l'execution de la requète");
      }

      //parcours des données retournées
      affiche("parcours des données retournées");
      try {
         ResultSetMetaData rsmd = résultats.getMetaData();
         int nbCols = rsmd.getColumnCount();
         boolean encore = résultats.next();

         while (encore) {

            for (int i = 1; i <= nbCols; i++)
               System.out.print(résultats.getString(i) + " ");
            System.out.println();
            encore = résultats.next();
         }

         résultats.close();
      } catch (SQLException e) {
         arret(e.getMessage());
      }

      affiche("fin du programme");
      System.exit(0);
   }
}
```

résultat :

```
connection a la base de données
creation enregistrement
nb mise a jour = 1
creation et execution de la requète
parcours des données retournées
1.0 client 1 prenom 1
2.0 client 2 prenom 2
3.0 client 3 client 4
fin du programme
```

19.7. Obtenir des informations sur la base de données

19.7.1. La classe ResultSetMetaData

La méthode getMetaData() d'un objet ResultSet retourne un objet de typeResultSetMetaData. Cet objet permet de connaître le nombre, le nom et le types des colonnes

Méthode	Role

int getColumnCount()	retourne le nombre de colonnes du ResultSet
String getColumnName(int)	retourne le nom de la colonne dont le numéro est donné
String getColumnLabel(int)	retourne le libellé de la colonne donnée
boolean isCurrency(int)	retourne true si la colonne contient un nombre au format monétaire
boolean isReadOnly(int)	retourne true si la colonne est en lecture seule
boolean isAutoIncrement(int)	retourne true si la colonne est auto incrémentée
int getColumnType(int)	retourne le type de données SQL de la colonne

19.7.2 La classe DatabaseMetaData

Un objet de la classe DatabaseMetaData permet d'obtenir des informations sur la base de données dans son ensemble : nom des tables, nom des colonnes dans une table, méthodes SQL supportées

Méthode	Role
ResultSet getCatalogs()	retourne la liste du catalogue d'informations (Avec le pont JDBC−ODBC, on obtient la liste des bases de données enregistrées dans ODBC).
ResultSet getTables(catalog, schema, tableNames, columnNames)	retourne une description de toutes les tables correspondant au TableNames donné et à toutes les colonnes correspondantes à columnNames).
ResultSet getColumns(catalog, schema, tableNames, columnNames)	retourne une description de toutes les colonnes correspondantes au TableNames donné et à toutes les colonnes correspondantes à columnNames).
String getURL()	retourne l'URL de la base à laquelle on est connecté
String getDriverName()	retourne le nom du driver utilisé

La méthode getTables() de l'objet DataBaseMetaData demande quatre arguments :

getTables(catalog, schema, tablemask, types[]);

- catalog : le nom du catalogue dans lequel les tables doivent être recherchées. Pour une base de données JDBC−ODBC, il peut être mis à null.
- schema : le schéma de la base données à inclure pour les bases les supportant. Il est en principe mis à null
- tablemask : un masque décrivant les noms des tables à retrouver. Pour les retrouver toutes, il faut l'initialiser avec la caractères ‘%'
- types[] : tableau de chaines décrivant le type de tables à retrouver. La valeur null permet de retrouver toutes les tables.

Exemple (code java 1.1) :

```
con = DriverManager.getConnection(url);
dma =con.getMetaData();
String[] types = new String[1];
types[0] = "TABLES"; //set table type mask

results = dma.getTables(null, null, "%", types);

boolean more = results.next();
while (more) {
```

```
   for (i = 1; i <= numCols; i++)
      System.out.print(results.getString(i)+" ");
   System.out.println();
   more = results.next();
}
```

19.8. L'utilisation d'un objet PreparedStatement

Cette section est en cours d'écriture

19.9. L'utilisation des transactions

Une transaction permet de ne valider un ensemble de traitements sur le base de données que si ils se sont tous effectués correctement.

Par exemple, une opération bancaire de transfert de fond d'un compte vers un autre oblige à la réalisation de l'opération de débit sur un compte et de l'opération de crédit sur l'autre compte. La réalisation d'une seule de ces opérations laisserait la base de données dans un état inconsistant.

Une transaction est un mécanisme qui permet de s'assurer que toutes les opérations qui la compose seront réellement effectuées.

Une transaction est gérée à partir de l'objet Connection. Par défaut, une connection est en mode auto−commit. Dans ce mode, chaque opération est validée unitairement pour formé la transaction.

Pour pouvoir rassembler plusieurs traitements dans une transaction, il faut tout d'abord désactiver le mode auto−commit. La classe Connection possède la méthode setAutoCommit() qui attend un boolean qui précise le mode fonctionnement.

Exemple :

connection.setAutoCommit(false);

Une fois le mode auto−commit désactivé, un appel à la méthode commit() de la classe Connection permet de valider la transaction courante. L'appel à cette méthode valide la transaction courante et créé implicitement une nouvelle transaction.

Si une anomalie intervient durant la transaction, il est possible de faire un retour en arrière pour revenir à la situation de la base de données au début de la transaction en appellant la méthode rollback() de la classe Connection.

19.10. Les procédures stockées

Cette section est en cours d'écriture

19.11. Le traitement des erreurs JDBC

JDBC permet de voir les avertissements et les exceptions générées par la base de données.

La classe SQLException représente les erreurs émises par la base de données. Elle contient trois attributs qui permettent de préciser l'erreur :

- message : contient une description de l'erreur
- SQLState : code définie par la norme ANSI 92
- ErrorCode : le code d'erreur du fournisseur du pilote

La classe SQLException possède une méthode getNextException() qui permet d'obtenir les autres exceptions levées durant la requête. La méthode renvoie null une fois la dernière exception renvoyée.

Cette section est en cours d'écriture

19.12. JDBC 2.0

La version 2.0 de l'API JDBC a été intégrée au JDK 1.2. Cette nouvelle version apporte plusieurs fonctionnalités très intéréssantes dont les principales sont :

- support du parcours dans les deux sens des résultats
- support de la mise à jour des résultats
- possibilité de faire des mise à jour de masse (Batch Updates)
- prise en compte des champs défini par SQL−3 dont BLOB et CLOB

JDBC 2.0 est séparé en deux parties :

- la partie principale (core API) contient les classes et interfaces nécessaires à l'utilisation de bases de données : elles sont regroupées dans le package java.sql
- la seconde partie est une extension utilisée dans J2EE qui permet de gérer les transactions distribuées, les pools de connection, la connection avec un objet DataSource ... Les classes et interfaces sont regroupées dans le package javax.sql

19.12.1. Les fonctionnalités de l'objet ResultSet

Les possibilités de l'objet ResultSet dans le version 1.0 de JDBC sont très limitées : parcours séquentiel de chaque occurence de la table retournée.

La version 2.0 apporte de nombreuses améliorations à cet objet : le parcours des occurences dans les deux sens et la possibilité de faire des mises à jour sur une occurence.

Concernant le parcours, il est possible de préciser trois mode des fonctionnement :

- forward−only : parcours sequentiel de chaque occurence (java.sql.ResultSet.TYPE_FORWARD_ONLY)
- scroll−insensitive : les occurences ne refletent pas les mises à jour qui peuvent intervenir durant le parcours (java.sql.ResultSet.TYPE_SCROLL_INSENSITIVE)
- scroll−sensitive : les occurences refletent les mises à jour qui peiuvent intervenir durant le parcours (java.sql.ResultSet.TYPE_SCROLL_SENSITIVE)

Il est aussi possible de préciser si le ResultSet peut être mise à jour ou non :

- java.sql.ResultSet.CONCUR_READ_ONLY : lecture seule
- java.sql.resultSet.CONCUR_UPDATABLE : mise à jour

C'est la création d'un objet de type Statement qu'il faut préciser ces deux modes. Si aucun des deux modes n'est précisé, ce sont les caractéristiques de la version 1.0 de JDBC qui sont utilisés (TYPE_FORWARD_ONLY et CONCUR_READ_ONLY).

Exemple (code jdbc 2.0) :

```
Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_REA
ResultSet resultSet = statement.executeQuery("SELECT nom, prenom FROM employes");
```

Le suuport de ces fonctionnalités est optionnel pour un pilote. L'objet DatabaseMetadata possède la méthode supportsResultSetType() qui attend en paramètre u constante sui représente une caractéristique : la méthode renvoie un booléen qui indique si la caractéristique est supportée ou non.

A la création du ResultSet, le curseur pour son parcours est positionné avant la première occurence à traiter. Pour sedéplacer dans l'ensemble des occurences, il y a toujours la méthode next() pour se déplacer sur le suivant mais aussi plusieurs autres méthodes pour permettre le parcours des occurences en fonctions du mode utilisé dont les principales sont :

Méthode	Rôle
boolean isBeforeFirst()	booleen qui indique si la position courante du curseur se trouve avant la première ligne
boolean isAfterLast()	booleen qui indique si la position courante du curseur se trouve après la dernière ligne
boolean isFirst()	booleen qui indique si le curseur est positionné sur la première ligne
boolean isLast()	booleen qui indique si le curseur est positionné sur la dernière ligne
boolean first()	déplacer le curseur sur la première ligne
boolean last()	déplacer le curseur sur la dernière ligne
boolean absolute()	déplace le curseur sur la ligne dont le numéro est fournie en paramètre à partir du début si il est positif et à partir de la fin si il est négatif. 1 déplace sur la première ligne, −1 sur la dernière, −2 sur l'avant dernière ...
boolean relative(int)	déplacer le curseur du nombre de lignes fourni en paramètre par rapport à la position courante du curseur. Le paramètre doit être négatif pour se déplacer vers le début et positif pur se déplacer vers la fin. Avant l'appel de cette méthode, il faut obligatoirement que le curseur soit positionné sur une ligne.
boolean previous()	déplacer le curseur sur la ligne précédente. Le booleen indique si la première occurence est dépassée.
void afterLast()	déplacer le curseur après la dernière ligne
void beforeFirst()	deplacer le curseur avant la première ligne

int getRow()	renvoie le numero de la ligne courante

Exemple (code jdbc 2.0) :

```
Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_REA
ResultSet resultSet = statement.executeQuery("SELECT nom, prenom FROM employes ORDER BY nom");
resultSet.afterLast();
while (resultSet.previous()) {
  System.out.println(resultSet.getString("nom")+" "+resultSet.getString("prenom"));
}
```

Durant le parcours d'un ResultSet, il est possible d'effectuer des mises à jour sur la ligne courante du curseur. Pour cela, il faut déclarer l'objet ResultSet comme acceptant les mise à jour. Avec les versions précédentes de JDBC, il fallait utiliser la méthode executeUpdate() avec une requète SQL.

Maitenant pour réaliser ces mises à jour, JDBC 2.0 propose de les réaliser via des appels de méthodes plutôt que d'utiliser des requêtes SQL.

Méthode	Rôle
updateXXX(String, XXX)	permet de mettre à jour la colonne dont le nom est fourni en paramètre. Le type java de cette colonne est XXX
updateXXX(int, XXX)	permet de mettre à jour la colonne dont l'index est fourni en paramètre. Le type java de cette colonne est XXX
updateRow()	permet d'actualiser les modifications réalisées avec des appels à updateXXX()
boolean rowsUpdated()	indique si la ligne courante aété modifiée
deleteRow()	Supprimer la ligne courante
rowDeleted()	indique si la ligne courante est supprimée
moveToInsertRow()	permet de créer une nouvelle ligne dans l'ensemble de résultat
inserRow()	permet de valider la création de la ligne

Pour réaliser une mise à jour dans la ligne courante désignée par le curseur, il faut utiliser une des méthodes updateXXX() sur chacun des champs à modifier. Une fois toutes les modifications faites dans une ligne, il faut appeler la méthode updateRow() pour reporter ces modifications dans la base de données car les méthodes updateXXX() ne font des mises à jour que dans le jeu de résultats. Les mises à jour sont perdues si un changement de ligne intervient avant l'appel à la méthode updateRow().

La méthode cancelRowUpdates() permet d'annuler toutes les modifications faites dans la ligne. L'appel à cette méthode doit être effectué avant l'appel à la méthode updateRow().

Pour insérer une nouvelle ligne dans le jeu de résultat, il faut tout d'abord appeler la méthode moveToInsertRow(). Cette méthode déplace le curseur vers un buffer dédié à la création d'une nouvelle ligne. Il faut alimenter chacun des champs nécessaires dans cette nouvelle ligne. Pour valider la création de cette nouvelle ligne, il faut appeler la méthode insertRow().

Pour supprimer la ligne courante, il faut appeler la méthode deleteRow(). Cette méthode agit sur le jeu de résultats et sur la base de données.

19.12.2. Les mises à jour de masse (Batch Updates)

JDBC 2.0 permet de réaliser des mises à jour de masse en regroupant plusieurs traitements pour les envoyer en une seule fois au SGBD. Ceci permet d'améliorer les performances surtout si le nombre de traitements est important.

Cette fonctionnalité n'est pas obligatoirement supportée par le pilote. La méthode supportsBatchUpdate() de la classe DatabaseMetaData permet de savoir si elle est utilisable avec le pilote.

Plusieurs méthodes ont été ajoutées à l'interface Statement pour pouvoir utiliser les mises à jour de masse :

Méthode	Rôle
void addBatch(String)	Permet d'ajouter une chaîne contenant une requete SQL
int[] executeBatch()	Cette méthode permet d'éxécuter toutes les requètes. Ell renvoit un tableau d'entier qui contient pour chaque requète, le nombre de mises à jour affectuées.
void clearBatch()	Supprimer toutes les requètes stockées

Lors de l'utilisation de batchupdate, il est préférable de positionner l'attribut autocommit à false afin de faciliter la gestion des transactions et le traitement d'une erreur dans l'execution d'un ou plusieurs traitements.

Exemple (code jdbc 2.0) :

```
connection.setAutoCommit(false);
Statement statement = connection.createStatement();

for(int i=0; i<10 ; i++) {
   statement.addBatch("INSERT INTO personne VALUES('nom"+i+"','prenom"+i+"')");
}
statement.executeBatch();
```

Une exception particulière peut être levée en plus de l'exception SQLException lors de l'éxécution d'une mise à jour de masse. L'exception SQLException est levée si une requète SQL d'intérogation doit être éxécutée (requête de type SELECT). L'exception BatchUpdateException est levée si une des requêtes de mise à jour échoue.

L'exception BatchUpdateException possède une méthode getUpdateCounts() qui renvoie un tableau d'entier qui contient le nombre d'occurence impacté par chaque requète reussie.

19.12.3. Le package javax.sql

Ce package est une extension à l'API JDBC qui propose des fonctionnalités pour les développements coté serveur. C'est pour cette raison, que cette extension est uniquement intégré à J2EE.

Les principales fonctionnalités proposées sont :

- une nouvelle interface pour assurer la connection : l'interface DataSource
- les pools de connections
- les transactions distribuées

- l'API Rowset

DataSource et Rowset peuvent être utilisé directement. Les pools de connections et les transactions distribuées sont utilisés par les serveurs d'applications pour fournir ces services.

19.12.4. La classe DataSource

La classe DataSource propose de fournir une meilleure alternative à la classe DriverManager pour assurer la connection à une base de données.

Elle représente une connection physique à une base de données. Les fournisseurs de pilotes proposent une implémentation de l'interface DataSource.

L'utilisation d'un objet DataSource est obligatoire pour pouvoir utiliser un pools de connection et les transactions distribuées.

Cette section est en cours d'écriture

19.12.5. Les pools de connection

Un pool de connections permet de maintenir un ensemble de connections établies vers une base de données qui sont réutilisables. L'établissement d'une connection est très couteux. L'intérêt du pool de connections est de limiter le nombre de ces créations et ainsi d'améliorer les performances surtout si le nombre de connections est important.

Cette section est en cours d'écriture

19.12.6. Les transactions distribuées

Les connections obtenues à partir d'un objet DataSource peuvent participer à une transactions distribuées.

Cette section est en cours d'écriture

19.12.7. L'API RowSet

Cette section est en cours d'écriture

19.13. JDBC 3.0

La version 3.0 de l'API JDBC a été intégrée au JDK 1.4 SE.

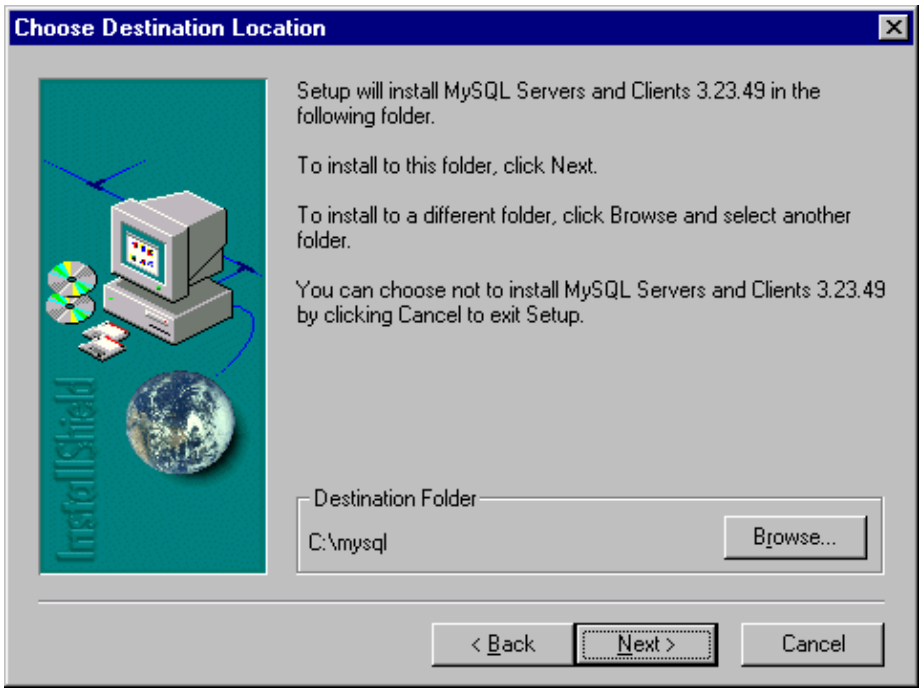
Cette section est en cours d'écriture

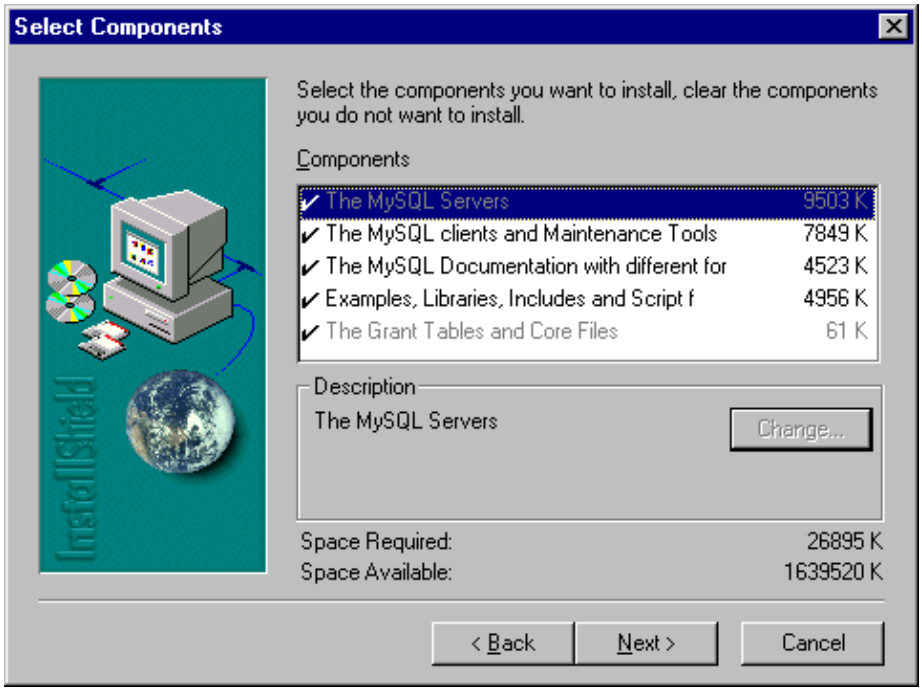
19.14. MySQL et Java

MySQL est une des bases de données open source les plus populaire.

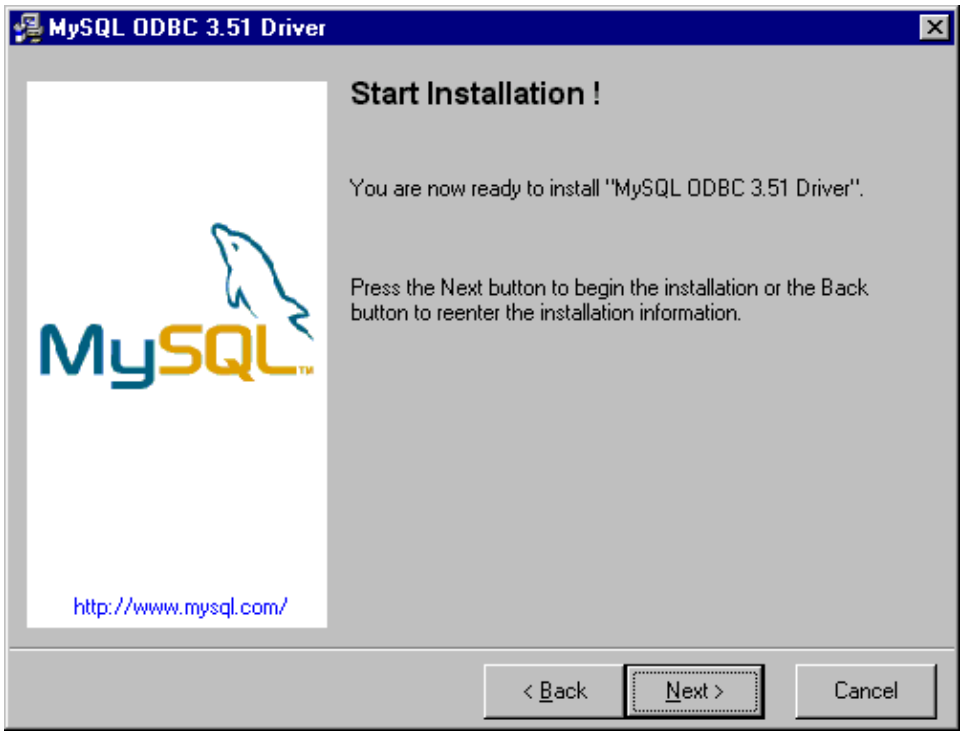
19.14.1. Installation sous windows

Il suffit de faire un download du fichier mysql−3.23.49−win.zip sur le site www.mysql.com, de decompresser le fichier dans un repertoire et d'éxécuter le fichier setup.exe





Il faut ensuite downloader le pilote ODBC, MyODBC−3.51.03.exe, et l'éxécuter



19.14.2. Utilisation de MySQL

Cette section est une présentation rapide de quelques fonctionnalités de base pour pouvoir utiliser MySQL. Pour un complément d'information sur toutes les possibilités de MySQL, consulter la documentation de cet excellent outils.

S'assurer que le serveur est lancé sinon executer la command c:\mysql\bin\mysqld−max

Pour executer des commandes SQL, il faut utiliser l'outils c:\mysql\bin\mysql. Cet outils est un interpreteur de commandes.

Exemple : pour voir les databases existantes

```
mysql>show databases;
+----------+
| Database |
+----------+
| mysql    |
| test     |
+----------+
2 rows in set (0.00 sec)
```

Un des premières choses à faire, c'est de créer une base de données qui va recevoir les différentes tables.

Exemple : Pour créer une nouvelle base de données nommée ‘testjava’

```
mysql> create database testjava;
Query OK, 1 row affected (0.00 sec)

mysql>use testjava;
Database changed
```

Cette nouvelle, base de données ne contient aucune table. Il faut créer la ou les tables utiles aux développements.

Exemple : Création d'une table nommée personne contenant trois champs : nom, prenom et date de naissance

```
mysql> show tables;
Empty set (0.06 sec)

mysql> create table personne (nom varchar(30), prenom varchar(30), datenais date
);
Query OK, 0 rows affected (0.00 sec)

mysql>show tables;
+--------------------+
| Tables_in_testjava |
+--------------------+
| personne           |
+--------------------+
1 row in set (0.00 sec)
```

Pour voir la définition de la table il faut utiliser la commande DESCRIBE :

Exemple : voir la définition de la table

```
mysql> describe personne;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| nom      | varchar(30) | YES  |     | NULL    |       |
| prenom   | varchar(30) | YES  |     | NULL    |       |
| datenais | date        | YES  |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

Cette table ne contient aucun enregistrement. Pour ajouter un enregistrement, il faut utiliser la command SQL insert.

Exemple : insertion d'une ligne dans la table

```
mysql> select * from personne;
Empty set (0.00 sec)

mysql> insert into personne values ('Nom 1','Prenom 1','1970-08-11');
Query OK, 1 row affected (0.05 sec)
```

```
mysql> select * from personne;
+-------+----------+------------+
| nom   | prenom   | datenais   |
+-------+----------+------------+
| Nom 1 | Prenom 1 | 1970-08-11 |
+-------+----------+------------+
1 row in set (0.00 sec)
```

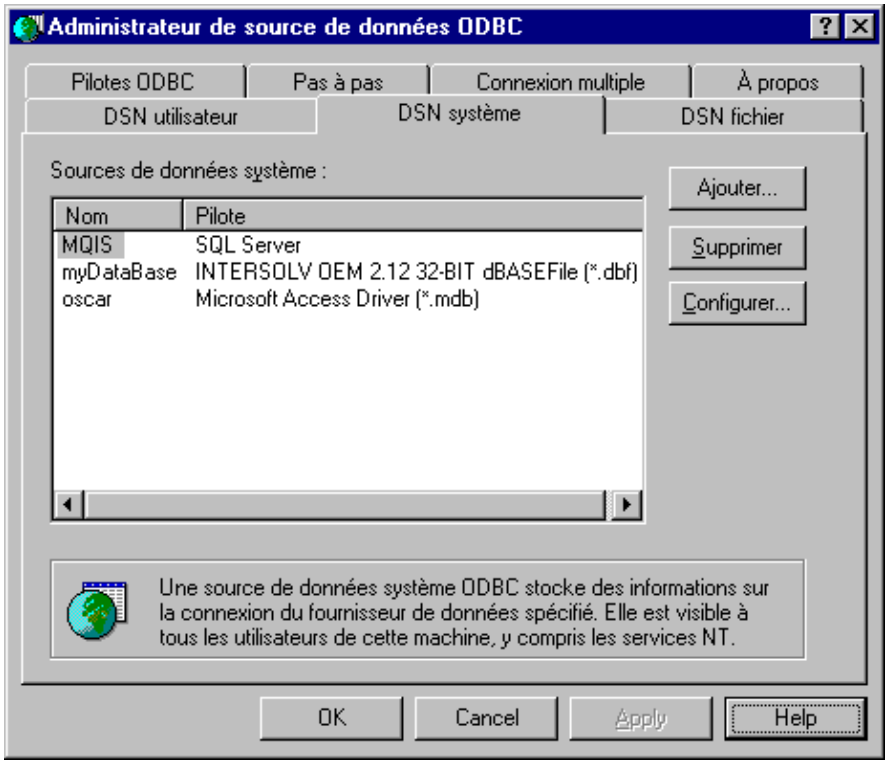
Il est existe des outils graphiques libres ou commerciaux pour faciliter l'administration et l'utilisation de MySQL.

19.14.3. Utilisation de MySQL avec java via ODBC

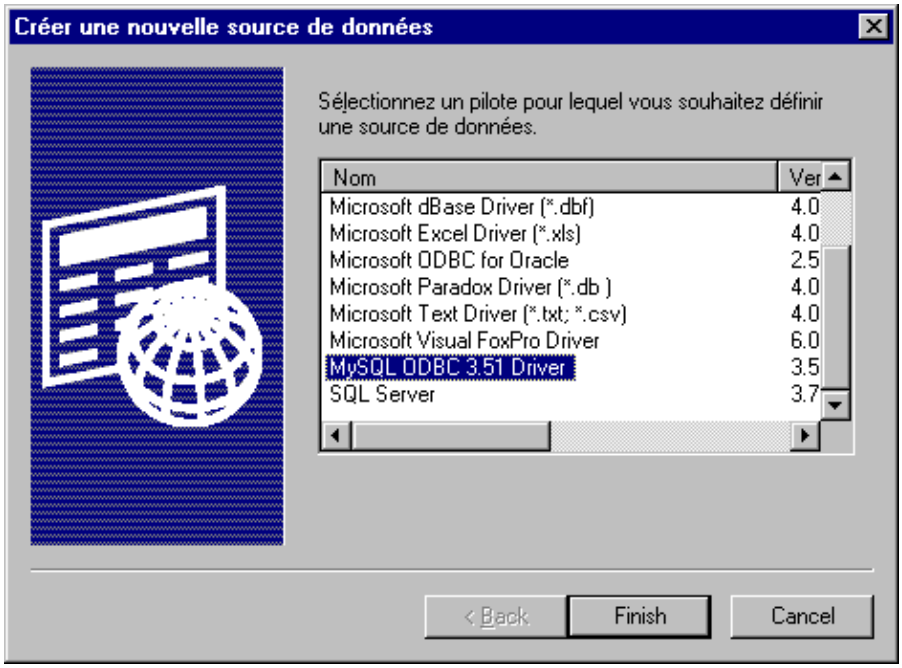
19.14.3.1. Déclaration d'une source de données ODBC vers la base de données

Dans le panneau de configuration, cliquer sur l'icône « Source de données ODBC ».

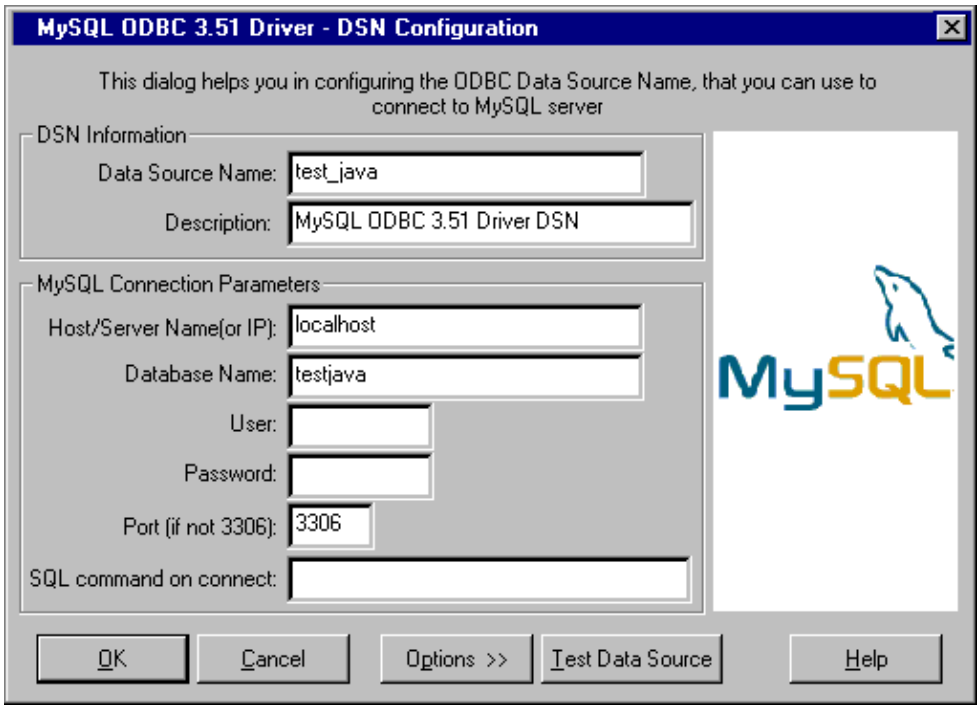
Le plus simple est de créer une source de données Systeme qui pourra être utilée par tous les utlisateurs en cliquant sur l'onglet DSN systeme



Pour ajouter une nouvelle source de données, il suffit de cliquer sur le bouton "Ajouter". Une boîte de dialogue permet de sélectionner le type de pilote qui sera utilisé par la source de données.

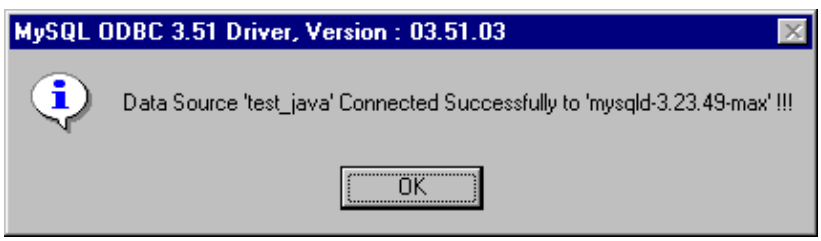


Il faut sélectionner le pilote mySQL et cliquer sur le bouton "Finish".

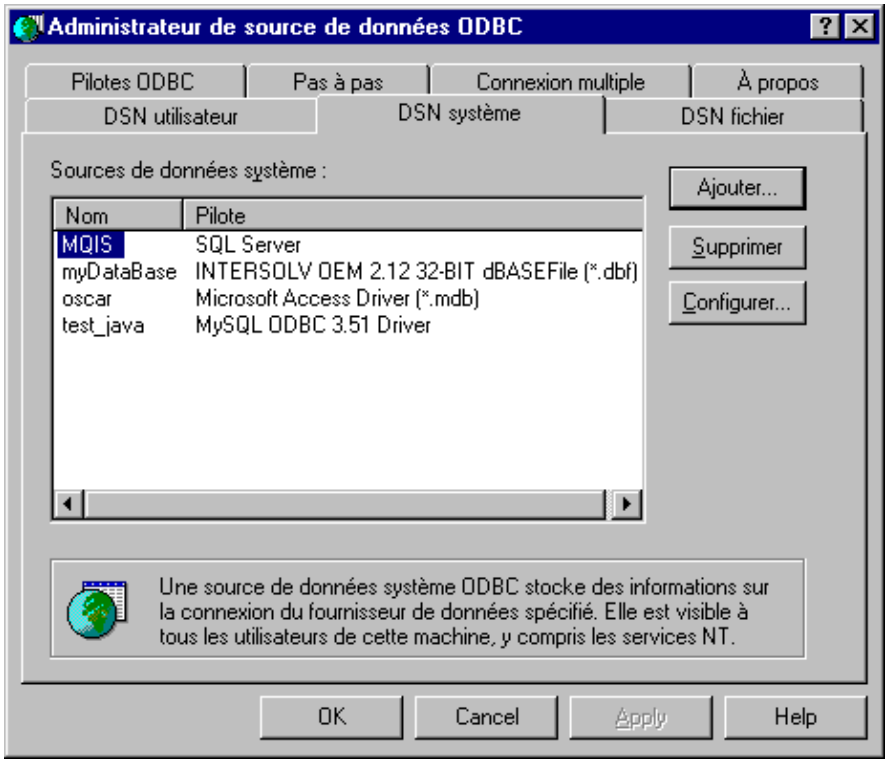


Une nouvelle boîte de dialogue permet de renseigner les informations sur la base de données à utiliser notamment le nom de DSN et le nom de la base de données.

Pour verifier si la connection est possible, il suffit de cliquer sur le bouton « Test Data Source »



Cliquer sur Ok pour fermer la fenetre et cliquer sur Ok pour valider les paramètres et créer la source de données.



La source de données est créée.

19.14.3.2. Utilisation de la source de données

Pour utiliser la source de données, il faut écrire et tester une classe java

Exemple

```
import java.sql.*;

public class TestJDBC10 {

  private static void affiche(String message) {
    System.out.println(message);
  }

  private static void arret(String message) {
    System.err.println(message);
    System.exit(99);
  }

  public static void main(java.lang.String[] args) {
    Connection con = null;
    ResultSet resultats = null;
    String requete = "";

     // chargement du pilote
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    } catch (ClassNotFoundException e) {
      arret("Impossible de charger le pilote jdbc:odbc");
    }

    //connection a la base de données
    affiche("connection a la base de donnees");
    try {

      String DBurl = "jdbc:odbc:test_java";
      con = DriverManager.getConnection(DBurl);
    } catch (SQLException e) {
      arret("Connection à la base de donnees impossible");
    }
```

```
    //creation et execution de la requète
    affiche("creation et execution de la requète");
    requete = "SELECT * FROM personne";

    try {
      Statement stmt = con.createStatement();
      resultats = stmt.executeQuery(requete);
    } catch (SQLException e) {
      arret("Anomalie lors de l'execution de la requète");
    }

    //parcours des données retournees
    affiche("parcours des données retournees");
    try {
      ResultSetMetaData rsmd = resultats.getMetaData();
      int nbCols = rsmd.getColumnCount();
      boolean encore = resultats.next();

      while (encore) {

        for (int i = 1; i <= nbCols; i++)
          System.out.print(resultats.getString(i) + "");

        System.out.println();

        encore = resultats.next();
      }

      resultats.close();
    } catch (SQLException e) {
      arret(e.getMessage());
    }

    affiche("fin du programme");
    System.exit(0);
  }
}
```

Resultat :

```
C:\$user>javac TestJDBC10.java
C:\$user>java TestJDBC10
connection a la base de donnees
creation et execution de la requ_te
parcours des donn_es retournees
Nom 1 Prenom 1 1970-08-11
fin du programme
```

19.14.4. Utilisation de MySQL avec java via un pilote JDBC

mm.mysql est un pilote JDBC de type IV developpé sous licence LGPL par Mark Matthews pour accéder à une base de données MySQL.

Le download du pilote JDBC se fait sur le site http://mmmysql.sourceforge.net/ . Le fichier mm.mysql−2.0.14−you−must−unjar−me.jar contient les sources et les binaires du pilote.

Pour utiliser cette archive, il faut la décompresser, par exemple dans le répertoire d'installation de mysql.

S'assurer que les fichiers jar sont accessibles dans le classpath ou les préciser manuellement lors de la compilation et de l'execution comme dans l'exemple ci dessous.

Exemple

```
import java.sql.*;

public class TestJDBC11 {
```

```
  private static void affiche(String message) {
    System.out.println(message);
  }

  private static void arret(String message) {
    System.err.println(message);
    System.exit(99);
  }

  public static void main(java.lang.String[] args) {
    Connection con = null;
    ResultSetresultats = null;
    String requete = "";

    // chargement du pilote
    try {
      Class.forName("org.gjt.mm.mysql.Driver").newInstance();
    } catch (Exception e) {
      arret("Impossible decharger le pilote jdbc pour mySQL");
    }

    //connection a la base de données
    affiche("connection a la base de donnees");
    try {

      String DBurl = "jdbc:mysql://localhost/testjava";
      con = DriverManager.getConnection(DBurl);
    } catch (SQLException e) {
      arret("Connection a la base de donnees impossible");
    }

    //creation et execution de la requète
    affiche("creation et execution dela requète");
    requete = "SELECT * FROM personne";

    try {
      Statement stmt = con.createStatement();
      resultats = stmt.executeQuery(requete);
    } catch (SQLException e) {
      arret("Anomalie lors de l'execution de la requete");
    }

    //parcours des données retournees
    affiche("Parcours des donnees retournees");
    try {
      ResultSetMetaData rsmd = resultats.getMetaData();
      int nbCols = rsmd.getColumnCount();
      boolean encore = resultats.next();

      while (encore) {

        for (int i = 1; i <= nbCols; i++)
          System.out.print(resultats.getString(i) + "");

        System.out.println();
        encore = resultats.next();
      }

      resultats.close();
    } catch (SQLException e) {
      arret(e.getMessage());
    }

    affiche("fin du programme");
    System.exit(0);
  }
}
```

Le programme est identique au précédent utilisant ODBC sauf :

- le nom de la classe du pilote

- l’URL de connectionà la base qui dépend du pilote

Resultat :

```
C:\$user>javac −classpath c:\j2sdk1.4.0−rc\jre\lib\mm.mysql−2.0.14−bin.jar TestJDBC11.java
C:\$user>
C:\$user>java −cp .;c:\j2sdk1.4.0−rc\jre\lib\mm.mysql−2.0.14−bin.jar TestJDBC11
connection a la base de donnees
creation et execution de la requ_te
Parcours des donnees retournees
Nom 1 Prenom 1 1970−08−11
fin du programme
```

20. La gestion dynamique des objets et l'introspection

Depuis la version 1.1 de java, il est possible de créer et de gérer dynamiquement des objets.

L'introspection est un mécanisme qui permet de connaître le contenu d'une classe dynamiquement. Il permet notamment de savoir ce que contient une classe sans en avoir les sources. Ces mécanismes sont largement utilisés dans des outils de type IDE (Integrated Development Environnement : environnement de développement intégré).

Pour illustrer ces différents mécanismes, ce chapitre va construire une classe qui proposera un ensemble de méthodes pour obtenir des informations sur une classe.

Les différentes classes utiles pour l'introspection sont rassemblées dans le package java.lang.reflect.

Voici le début de cette classe qui attend dans son constructeur une chaîne de caractères précisant la classe sur laquelle elle va travailler.

Exemple (code java 1.1) :

```
import java.util.*;
import java.lang.reflect.*;
public class ClasseInspecteur {
  private Class classe;
  private String nomClasse;
  public ClasseInspecteur(String nomClasse) {
    this.nomClasse = nomClasse;
    try {
      classe = Class.forName(nomClasse);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

20.1. La classe Class

Les instances de la classe Class sont des objets représentant les classes du langage. Il y aura une instance représentant chaque classes utilisées : par exemple la classe String, la classe Frame, la classe Class, etc ... . Ces instances sont crées automatiquement par la machine virtuelle lors du chargement de la classe. Il est ainsi possible de connaître les caractéristiques d'une classe de façon dynamique en utilisant les méthodes de la classe Class. Les applications telles que les debuggers, les inspecteurs d'objets et les environnement de développement doivent faire une analyse des objets qu'ils manipulent en utilisant ces mécanismes.

La classe Class est définie dans le package java.lang.

La classe Class permet :

- de décrire une classe ou une interface par introspection : obtenir son nom, sa classe mère, la liste de ces méthodes, de ses variables de classe, de ses constructeurs et variables d'instances, etc ...

- d'agir sur une classe en envoyant, à un objet Class des messages comme à tout autre objet. Par exemple, créer dynamiquement à partir d'un objet Class une nouvelle instance de la classe représentée

20.1.1. Obtenir un objet de la classe Class

La classe Class ne possède pas de constructeur public mais il existe plusieurs façons d'obtenir un objet de la classe Class.

20.1.1.1. Connaître la classe d'un objet

La méthode getClass() définit dans la classe Object renvoie une instance de la classe Class. Par héritage, tout objet java dispose de cette méthode.

Exemple (code java 1.1) :

```
package introspection;

public class TestGetClass {
   public static void main(java.lang.String[] args) {
      String chaine = "test";
      Class classe = chaine.getClass();
      System.out.println("classe de l'objet chaine = "+classe.getName());
   }
}
```

Résultat :

```
classe de l'objet chaine = java.lang.String
```

20.1.1.2. Obtenir un objet Class à partir d'un nom de classe

La classe Class possède une méthode statique forName() qui permet à partir d'une chaîne de caractères désignant une classe d'instancier un objet de cette classe et de renvoyer un objet de la classe Class pour cette classe.

Cette méthode peut lever l'exception ClassNotFoundException.

Exemple (code java 1.1) :

```
public class TestForName {
   public static void main(java.lang.String[] args) {
      try {
         Class classe = Class.forName("java.lang.String");
         System.out.println("classe de l'objet chaine = "+classe.getName());
      }
      catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

Résultat :

```
classe de l'objet chaîne = java.lang.String
```

20.1.1.3. Une troisième façon d'obtenir un objet Class

Il est possible d'avoir un objet de la classe Class en écrivant type.class ou type est le nom d'une classe.

Exemple (code java 1.1) :

```
package introspection;
public class TestClass {
   public static void main(java.lang.String[] args) {
      Class c = Object.class;
      System.out.println("classe de Object  = "+c.getName());
   }
}
```

Résultat :

```
classe de Object  = java.lang.Object
```

20.1.2. Les méthodes de la classe Class

La classe Class fournie de nombreuses méthodes pour obtenir des informations sur la classe qu'elle représente. Voici les principales méthodes :

Méthodes	Rôle
static Class forName(String)	Instancie un objet de la classe dont le nom est fourni en paramètre et renvoie un objet Class la représentant
Class[] getClasses()	Renvoie les classes et interfaces publiques qui sont membres de la classe
Constructor[] getConstructors()	Renvoie les constructeurs publics de la classe
Class[] getDeclaredClasses()	Renvoie un tableau des classes définies comme membre dans la classe
Constructor[] getDeclaredConstructors()	Renvoie tous les constructeurs de la classe
Field[] getDeclaredFields()	Renvoie un tableau de tous les attributs définis dans la classe
Method getDeclaredMethods()	Renvoie un tableau de toutes les méthodes
Field getFields()	Renvoie un tableau des attributs publics
Class[] getInterfaces()	Renvoie un tableau des interfaces implémentées par la classe
Method getMethod()	Renvoie un tableau des methodes publiques de la classe incluant celles héritées
int getModifiers()	Renvoie un entier qu'il faut décoder pour connaître les modificateurs de la classe
Package getPackage()	Renvoie le package de la classe
Classe getSuperClass()	Renvoie la classe mère de la classe
boolean isArray()	Indique si la classe est un tableau
boolean IsInterface()	Indique si la classe est une interface
Object newInstance()	Permet de créer une nouvelle instance de la classe

20.2. Rechercher des informations sur une classe

En utilisant les méthodes de la classe Class, il est possible d'obtenir quasiment toutes les informations sur une classe.

20.2.1. Rechercher la classe mère d'une classe

La classe Class possède une méthode getSuperClass() qui retourne un objet de la classe Class représentant la classe mère si elle existe sinon elle retourne null.

Pour obtenir toute la hiérarchie d'une classe il suffit d'appeler successivement cette méthode sur l'objet qu'elle a retourné.

Exemple (code java 1.1) : méthode qui retourne un vecteur contenant les classes mères

```
public Vector getClassesParentes() {

  Vector cp = new Vector();

  Class sousClasse = classe;
  Class superClasse;

  cp.add(sousClasse.getName());
  superClasse = sousClasse.getSuperclass();
  while (superClasse != null) {
    cp.add(0,superClasse.getName());
    sousClasse = superClasse;
    superClasse = sousClasse.getSuperclass();
  }
  return cp;
}
```

20.2.2. Rechercher les modificateurs d'une classe

La classe Class possède une méthode getModifiers() qui retourne un entier représentant les modificateurs de la classe. Pour décoder cette valeur, la classe Modifier possède plusieurs méthodes qui attendent cet entier en paramètre et qui retourne un booléen selon leur fonction : isPublic, isAbstract, isFinal ...

La classe Modifier ne contient que des constantes et des méthodes statiques qui permettent de déterminer les modificateurs d'accès :

Méthode	Rôle
boolean isAbstract(int)	Renvoie true si le paramètre contient le modificateur abstract
boolean isFinal(int)	Renvoie true si le paramètre contient le modificateur final
boolean isInterface(int)	Renvoie true si le paramètre contient le modificateur interface
boolean isNative(int)	Renvoie true si le paramètre contient le modificateur native
boolean isPrivate(int)	Renvoie true si le paramètre contient le modificateur private
boolean isProtected(int)	Renvoie true si le paramètre contient le modificateur protected
boolean isPublic(int)	Renvoie true si le paramètre contient le modificateur public
boolean isStatic(int)	Renvoie true si le paramètre contient le modificateur static
boolean isSynchronized(int)	Renvoie true si le paramètre contient le modificateur synchronized
boolean isTransient(int)	Renvoie true si le paramètre contient le modificateur transient
boolean isVolatile(int)	Renvoie true si le paramètre contient le modificateur volatile

Ces méthodes étant static il est inutile d'instancier un objet de type Modifier pour utiliser ces méthodes.

Exemple (code java 1.1) :

```
public Vector getModificateurs() {

  Vector cp = new Vector();
  int m = classe.getModifiers();
  if (Modifier.isPublic(m))
    cp.add("public");
  if (Modifier.isAbstract(m))
    cp.add("abstract");
  if (Modifier.isFinal(m))
    cp.add("final");

  return cp;
}
```

20.2.3. Rechercher les interfaces implémentées par une classe

La classe Class possède une méthode getInterfaces() qui retourne un tableau d'objet de type Class contenant les interfaces implémentées par la classe.

Exemple (code java 1.1) :

```
public Vector getInterfaces() {

  Vector cp = new Vector();

  Class[] interfaces = classe.getInterfaces();
  for (int i = 0; i < interfaces.length; i++) {
    cp.add(interfaces[i].getName());
  }

  return cp;
}
```

20.2.4. Rechercher les champs publics

La classe Class possède une méthode getFields() qui retourne les attributs public de la classe. Cette méthode retourne un tableau d'objet de type Field.

La classe Class possède aussi une méthode getField() qui attend en paramètre un nom d'attribut et retourne un objet de type Field si celui ci est défini dans la classe ou dans une de ses classes mères. Si la classe ne contient pas d'attribut dont le nom correspond au paramètre fourni, la méthode getField() lève une exception de la classe NoSuchFieldException.

La classe Field représente un attribut d'une classe ou d'une interface et permet d'obtenir des informations cet attribut. Elle possède plusieurs méthodes :

Méthode	Rôle
String getName()	Retourne le nom de l'attribut
Class getType()	Retourne un objet de type Class qui représente le type de l'attribut
Class getDeclaringClass()	Retourne un objet de type Class qui représente la classe qui définie l'attribut
int getModifiers()	Retourne un entier qui décrit les modificateurs d'accès. Pour les connaître précisément il faut utiliser les méthodes static de la classe Modifier.

Object get(Object)	Retourne la valeur de l'attribut pour l'instance de l'objet fourni en parameter. Il existe aussi plusieurs méthodes getXXX() ou XXX représente un type primitf et qui la renvoie la valeur dans ce type.

Exemple (code java 1.1) :

```
public Vector getChampsPublics() {

  Vector cp = new Vector();

  Field[] champs = classe.getFields();
  for (int i = 0; i < champs.length; i++)
    cp.add(champs[i].getType().getName()+" "+champs[i].getName());
  return cp;
}
```

20.2.5. Rechercher les paramètres d'une méthode ou d'un constructeurs

L'exemple ci dessous présente une méthode qui permet de formatter sous forme de chaîne de caractères les paramètres d'une méthode fourni sous la forme d'un tableau d'objets Class.

Exemple (code java 1.1) :

```
private String rechercheParametres(Class[] classes) {

  StringBuffer param = new StringBuffer("(");

  for (int i = 0; i < classes.length; i ++) {

    param.append(formatParametre(classes[i].getName()));
    if (i < classes.length - 1)
      param.append(", ");

  }
  param.append(")");

  return param.toString();

}
```

La méthode getName() de la classe Class renvoie une chaîne de caractères formatées qui précise le type de la classe.

Si le type de la classe est un tableau alors la chaîne commence par un nombre de caractère '[' correspondant à la dimension du tableau.

Ensuite la chaîne contient un caractère qui précise un type primitif ou un objet. Dans le cas d'un objet, le nom de la classe de l'objet avec son package complet est contenu dans la chaîne suivi d'un caractère ';'.

Caractère	Type
B	byte
C	char
D	double
F	float
I	int
J	long

L*classname;*	*classe ou interface*
S	short
Z	boolean

Exemple :

La méthode getName() de la classe Class représentant un objet de type float[10][5] renvoie « [[F »

Pour simplifier les traitements, la méthode formatParametre() ci dessous retourne une chaîne de caractères qui décode le contenu de la chaîne retournée par la méthode getName() de la classe Class.

Exemple (code java 1.1) :

```
  private String formatParametre(String s) {

    if (s.charAt(0) == '[') {

      StringBuffer param = new StringBuffer("");

      int dimension = 0;
      while (s.charAt(dimension) == '[') dimension++;

      switch(s.charAt(dimension)) {
        case 'B' : param.append("byte");break;
        case 'C' : param.append("char");break;
        case 'D' : param.append("double");break;
        case 'F' : param.append("float");break;
        case 'I' : param.append("int");break;
        case 'J' : param.append("long");break;
        case 'S' : param.append("short");break;
        case 'Z' : param.append("boolean");break;
        case 'L' : param.append(s.substring(dimension+1,s.indexOf(";")));
      }

      for (int i =0; i < dimension; i++)
        param.append("[]");

      return param.toString();
    }
    else return s;

  }
```

20.2.6. Rechercher les constructeurs de la classe

La classe Class possède une méthode getConstructors() qui retourne un tableau d'objet de type Constructor contenant les constructeurs de la classe.

La classe Constructor représente un constructeur d'une classe. Elle possède plusieurs méthodes :

Méthode	Rôle
String getName()	Retourne le nom du constructeur
Class[] getExceptionTypes()	Retourne un tableau de type Class qui représente les exceptions qui peuvent être propagées par le constructeur
Class[] getParametersType()	Retourne un tableau de type Class qui représente les paramètres du constructeur
int getModifiers()	

	Retourne un entier qui décrit les modificateurs d'accès. Pour les connaître précisément il faut utiliser les méthodes static de la classe Modifier.
Object newInstance(Object[])	Instancie un objet en utilisant le constructeur avec les paramètres fournis à la méthode

Exemple (code java 1.1) :

```
public Vector getConstructeurs() {

  Vector cp = new Vector();
  Constructor[] constructeurs = classe.getConstructors();
  for (int i = 0; i < constructeurs.length; i++) {
    cp.add(rechercheParametres(constructeurs[i].getParameterTypes()));

  }

  return cp;
}
```

L'exemple ci dessus utilise la méthode rechercherParamètres() définie précédemment pour simplifier les traitements.

20.2.7. Rechercher les méthodes publiques

Pour consulter les méthodes d'un objet, il faut obtenir sa classe et lui envoyer le message getMethod(), qui renvoie les méthodes publique qui sont déclarées dans la classe et qui sont héritées des classes mères.

Elle renvoie un tableau d'instances de la classe Method du package java.lang.reflect.

Une méthode est caractérisée par un nom, une valeur de retour, une liste de paramètres, une liste d'exceptions et une classe d'appartenance.

La classe Method contient plusieurs méthodes :

Méthode	Rôle
Class[] getParameterTypes	Renvoie un tableau de classes représentant les paramètres.
Class getReturnType	Renvoie le type de la valeur de retour de la méthode.
String getName()	Renvoie le nom de la méthode
int getModifiers()	Renvoie un entier qui représentent les modificateur d'accès
Class[] getExceptionTypes	Renvoie un tableau de classes contenant les exceptions propagées par la méthode
Class getDeclaringClass[]	Renvoie la classe qui définie la méthode

Exemple (code java 1.1) :

```
public Vector getMethodesPubliques() {

  Vector cp = new Vector();
  Method[] methodes = classe.getMethods();
  for (int i = 0; i < methodes.length; i++) {
    StringBuffer methode = new StringBuffer();

    methode.append(formatParametre(methodes[i].getReturnType().getName()));
    methode.append(" ");
    methode.append(methodes[i].getName());
    methode.append(rechercheParametres(methodes[i].getParameterTypes()));
```

```
    cp.add(methode.toString());

  }
  return cp;
}
```

L'exemple ci dessus utilise les méthodes formatParametre() et rechercherParamètres() définies précédemment pour simplifier les traitements.

20.2.8. Rechercher toutes les méthodes

Pour consulter toutes les méthodes d'un objet, il faut obtenir sa classe et lui envoyer le message getDeclareMethods(), qui renvoie toutes les méthodes qui sont déclarées dans la classe et qui sont héritées des classes mères quelque soit leur accessibilité.

Elle renvoie un tableau d'instances de la classe Method du package java.lang.reflect.

Exemple (code java 1.1) :

```
public Vector getMethodes() {

  Vector cp = new Vector();
  Method[] methodes = classe.getDeclareMethods();
  for (int i = 0; i < methodes.length; i++) {
    StringBuffer methode = new StringBuffer();

    methode.append(formatParametre(methodes[i].getReturnType().getName()));
    methode.append(" ");
    methode.append(methodes[i].getName());
    methode.append(rechercheParametres(methodes[i].getParameterTypes()));

    cp.add(methode.toString());

  }

  return cp;
}
```

L'exemple ci dessus utilise les méthodes formatParametre() et rechercherParamètres() définies précédemment pour simplifier les traitements.

20.3. Définir dynamiquement des objets

20.3.1. Définir des objets grâce à la classe Class

Cette section est en cours d'écriture

20.3.2. Exécuter dynamiquement une méthode

Cette section est en cours d'écriture

21. L'appel de méthodes distantes : RMI

RMI (Remote Method Invocation) est une technologie développée et fournie par Sun à partir du JDK 1.1 pour permettre de mettre en œuvre facilement des objets distribuées.

Ce chapitre contient plusieurs sections :

21.1. Présentation et architecture de RMI

Le but de RMI est de permettre l'appel, l'éxécution et le renvoi du résultat d'une méthode éxécuté dans une machine virtuelle différente de celle de l'objet l'appelant. Cette machine virtuelle peut être sur une machine différente pourvu qu'elle soit accessible par le réseau.

La machine sur laquelle s'éxécute la méthode distante est appelée serveur.

L'appel coté client d'une telle méthode est un peu plus compliqué que l'appel d'une méthode d'un objet local mais il reste simple. Il consiste à obtenir une référence sur l'objet distant puis à simplement appeler la méthode à partir de cette référence.

La technologie RMI se charge de rendre transparente la localisation de l'objet distant, son appel et le renvoi du résultat.

En fait, elle utilise deux classes particulières, le stub et le skeleton, qui doivent être générées avec l'outils rmic fourni avec le JDK.

Le stub est une classe qui se situe côté client et le skeleton est son homologue coté serveur. Ces deux classes se chargent d'assurer tous les mécanismes d'appel, de communication, d'execution, de renvoie et de réception du résultat.

21.2. Les différentes étapes pour créer un objet distant et l'appeler avec RMI

Le développement coté serveur se compose de :

- La définition d'une interface qui contient les méthodes qui peuvent être appelées à distance
- L'écriture d'une classe qui implémente cette interface

- L'écriture d'une classe qui instanciera l'objet et l'enregistrera en lui affectant un nom dans le registre de nom RMI (RMI Registry)

Le développement côté client se compose de :

- L'obtention d'une référence sur l'objet distant à partir de son nom
- L'appel à la méthode à partir de cette référence

Enfin, il faut générer les classes stub et skeleton en exécutant le programme rmic avec le fichier source de l'objet distant

21.3. Le développement coté serveur

21.3.1. La définition d'une interface qui contient les méthodes de l'objet distant

L'interface à définir doit hériter de l'interface java.rmi.Remote. Cette interface ne contient aucune méthode mais indique simplement que l'interface peut être appelée à distance.

L'interface doit contenir toutes les méthodes qui seront succeptibles d'être appelées à distance.

La communication entre le client et le serveur lors de l'invocation de la méthode distante peut échouer pour diverses raisons tel qu'un crash du serveur, une rupture de la liaison, etc …

Ainsi chaque méthode appelée à distance doit déclarer qu'elle est en mesure de lever l'exception java.rmi.RemoteException.

Exemple (code java 1.1) :

```
package test_rmi;

import java.rmi.*;

public interface Information extends Remote {

   public String getInformation() throws RemoteException;

}
```

21.3.2. L'écriture d'une classe qui implémente cette interface

Cette classe correspond à l'objet distant. Elle doit donc implémenter l'interface définie et contenir le code nécessaire.

Cette classe doit obligatoirement hériter de la classe UnicastRemoteObject qui contient les differents traitements élémentaires pour un objet distant dont l'appel par le stub du client est unique. Le stub ne peut obtenir qu'une seule référence sur un objet distant héritant de UnicastRemoteObject. On peut supposer qu'une future version de RMI sera capable de faire du MultiCast, permettant à RMI de choisir parmis plusieurs objets distants identiques la référence à fournir au client.

La hiérarchie de la classe UnicastRemoteObject est :

java.lang.Object

java.rmi.Server.RemoteObject

java.rmi.Server.RemoteServer

java.rmi.Server.UnicastRemoteObject

Comme indiqué dans l'interface, toutes les méthodes distantes doivent indiquer qu'elles peuvent lever l'exception RemoteException mais aussi le constructeur de la classe. Ainsi, même si le constructeur ne contient pas de code il doit être redéfini pour inhiber la génération du constructeur par défaut qui ne lève pas cette exception.

Exemple (code java 1.1) :

```
package test_rmi;

import java.rmi.*;

import java.rmi.server.*;

public class TestRMIServer extends UnicastRemoteObject implements Information {

   protected TestRMIServer() throws RemoteException {
      super();
   }

   public String getInformation()throws RemoteException {
      return "bonjour";
   }

}
```

21.3.3. L'écriture d'une classe pour instancier l'objet et l'enregistrer dans le registre

Ces opérations peuvent être effectuées dans la méthode main d'une classe dédiée ou dans la méthode main de la classe de l'objet distant. L'intérêt d'une classe dédié et qu'elle permet de regrouper toutes ces opérations pour un ensemble d'objets distants.

La marche a suivre contient trois étapes :

- la mise en place d'un security manager dédié qui est facultative
- l'instanciation d'un objet de la classe distante
- l'enregistrement de la classe dans le registre de nom RMI en lui donnant un nom

21.3.3.1. La mise en place d'un security manager

Cette opération n'est pas obligatoire mais elle est recommandée en particulier si le serveur doit charger des classes qui ne sont pas sur le serveur. Sans sécurity mananger, il faut obligatoirement mettre à la disposition du serveur toutes les classes dont il aura besoin (Elles doivent être dans le CLASSPATH du serveur). Avec un security manager, le serveur peut charger dynamiquement certaines classes.

Cependant, le chargement dynamique de ces classes peut poser des problèmes de sécurité car le serveur va éxécuter du code d'une autre machine. Cet aspect peut conduire à ne pas utiliser de sécurity manager.

Exemple (code java 1.1) :

```
public static void main(String[] args) {
   try {
      System.out.println("Mise en place du Security Manager ...");
      System.setSecurityManager(new java.rmi.RMISecurityManager());
   } catch (Exception e) {
      System.out.println("Exception capturée: " + e.getMessage());
   }
}
```

21.3.3.2. L'instanciation d'un objet de la classe distante

Cette opération est très simple puisqu'elle consiste simplement en la création d'un objet de la classe de l'objet distant

Exemple (code java 1.1) :

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

   try {
      System.out.println("Mise en place du Security Manager ...");
      System.setSecurityManager(new java.rmi.RMISecurityManager());

      TestRMIServer testRMIServer = new TestRMIServer();

   } catch (Exception e) {

      System.out.println("Exception capturée: " + e.getMessage());

   }

}
```

21.3.3.3. L'enregistrement dans le registre de nom RMI en lui donnant un nom

La dernière opération consiste à enregistrer l'objet créé dans le registre de nom en lui affectant un nom. Ce nom est fourni au registre sous forme d'une URL constitué du préfix rmi://, du nom du seveur (hostname) et du nom associé à l'objet précédé d'un slash.

Le nom du serveur peut être fourni « en dur » sous forme d'une constante chaine de caractères ou peut être dynamiquement obtenu en utilisant la classe InetAddress pour une utilisation en locale.

C'est ce nom qui sera utilisé dans une URL par le client pour obtenir une référence sur l'objet distant.

L'enregistrement se fait en utilisant la méthode rebind de la classe Naming. Elle attend en paramètre l'URL du nom de l'objet et l'objet lui même.

Exemple (code java 1.1) :

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

   try {

      System.out.println("Mise en place du Security Manager ...");
      System.setSecurityManager(new java.rmi.RMISecurityManager());

      TestRMIServer testRMIServer = new TestRMIServer();

      System.out.println("Enregistrement du serveur");

      Naming.rebind("rmi://"+java.net.InetAddress.getLocalHost()+
         "/TestRMI",testRMIServer);

      // Naming.rebind(";rmi://localhost/TestRMI", testRMIServer);

      System.out.println("Serveur lancé");

   } catch (Exception e) {
      System.out.println("Exception capturée: " + e.getMessage());
   }

}
```

21.3.3.4. Lancement dynamique du registre de nom RMI

Sur le serveur, le registre de nom RMI doit s'éxécuter avant de pouvoir enregistrer un objet ou obtenir une référence.

Ce registre peut être lancé en tant qu'application fournie par sun dans le JDK (rmiregistry) comme indiqué dans un chapitre suivant ou être lancé dynamiquement dans la classe qui enregistre l'objet. Ce lancement ne doit avoir lieu qu'une seule et unique fois. Il peut être intéressant d'utiliser ce code si l'on créé une classe dédié à l'enregistrement des objets distants.

Le code pour éxécuter le registre est la méthode createRegistry de la classe java.rmi.registry.LocateRegistry. Cette méthode attend en paramètre un numéro de port.

Exemple (code java 1.1) :

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

   try {

      java.rmi.registry.LocateRegistry.createRegistry(1099);

      System.out.println("Mise en place du Security Manager ...");
      System.setSecurityManager(new java.rmi.RMISecurityManager());

      ...

   }
}
```

21.4. Le développement coté client

L'appel d'une méthode distante peut se faire dans une application ou dans une applet.

21.4.1. La mise en place d'un security manager

Comme pour le coté serveur, cette opération est facultative.

Le choix de la mise en place d'un sécurity manager côté client suit des règles identiques à celui du côté serveur. Sans son utilisation, il est nécessaire de mettre dans le CLASSPATH du client toutes les classes nécessaires dont la classe stub.

Exemple (code java 1.1) :

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

   System.setSecurityManager(new RMISecurityManager());

}
```

21.4.2. L'obtension d'une référence sur l'objet distant à partir de son nom

Pour obtenir une référence sur l'objet distant à partir de son nom, il faut utiliser la méthode statique lookup() de la classe Naming.

Cette méthode attend en paramètre une URL indiquant le nom qui référence l'objet distant. Cette URL est composé de préfix rmi://, le nom du serveur (hostname) et le nom de l'objet tel qu'il a été enregistré dans le registre précédé d'un slash.

Il est préférable de prévoir le nom du serveur sous forme de paramètres de l'application ou de l'applet pour plus de souplesse.

La méthode lookup() va rechercher dans le registre du serveur l'objet et retourner un objet stub. L'objet retourné est de la classe Remote (cette classe est la classe mère de tous les objets distants).

Si le nom fourni dans l'URL n'est pas référencé dans le registre, la méthode lève l'exception NotBoundException.

Exemple (code java 1.1) :

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

   System.setSecurityManager(new RMISecurityManager());

   try {

      Remote r = Naming.lookup("rmi://vaio/127.0.0.1/TestRMI");

   } catch (Exception e) {
   }

}
```

21.4.3. L'appel à la méthode à partir de la référence sur l'objet distant

L'objet retourné étant de type Remote, il faut réaliser un cast vers l'interface qui définit les méthodes de l'objet distant. Pour plus de sécurité, on vérifie que l'objet retourné est bien une instance de cette interface.

Un fois le cast réalisé, il suffit simplement d'appeler la méthode.

Exemple (code java 1.1) :

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

   System.setSecurityManager(new RMISecurityManager());

   try {

      Remote r = Naming.lookup("rmi://vaio/127.0.0.1/TestRMI");

      if (r instanceof Information) {
         String s = ((Information) r).getInformation();
         System.out.println("chaine renvoyée = " + s);
      }

   } catch (Exception e) {
   }
}
```

21.4.4. L'appel d'une méthode distante dans une applet

L'appel d'une méthode distante est la même dans une application et dans une applet.

Seul la mise en place d'un security manager dédié dans les applets est inutile car elles utilisent déjà un sécurity manager (AppletSecurityManager) qui autorise le chargement de classes distantes.

Exemple (code java 1.1) :

```
package test_rmi;

import java.applet.*;
import java.awt.*;
import java.rmi.*;

public class AppletTestRMI extends Applet {

   private String s;
```

```
   public void init() {

      try {
         Remote r = Naming.lookup("rmi://vaio/127.0.0.1/TestRMI");

         if (r instanceof Information) {
            s = ((Information) r).getInformation();
         }
      } catch (Exception e) {
      }

   }

   public void paint(Graphics g) {
      super.paint(g);
      g.drawString("chaine retournée = "+s,20,20);
   }

}
```

21.5. La génération des classes stub et skeleton

Pour générer ces classes, il suffit d'utiliser l'outils rmic fourni avec le JDK en lui donnant en paramètre le nom de la classe.

Attention la classe doit avoir été compilée : rmic à besoin du fichier .class.

Exemple (code java 1.1) :
`rmic test_rmi.TestRMIServer`

rmic va générer et compiler les classes stub et skeleton respectivement sous le nom TestRMIServer_Stub.class et TestRMIServer_Skel.class

21.6. La mise en oeuvre des objets RMI

La mise en œuvre et l'utilisation d'objet distant avec RMI nécessite plusieurs étapes :

1. Démarrer le registre RMI sur le serveur soit en utilisant le programme rmiregistry livré avec le JDK soit en exécutant une classe qui éffectue le lancement.
2. éxécuter la classe qui instancie l'objet distant et l'enregistre dans le serveur de nom RMI
3. Lancer l'application ou l'applet pour tester.

21.6.1. Le lancement du registre RMI

La commande rmiregistry est fournie avec le JDK.

Il faut la lancer en tache de fond :

Sous Unix : rmiregistry&

Sous Windows : start rmiregistry

Ce registre permet de faire correspondre un objet à un nom et inversement. C'est lui qui est sollicité lors d'un appelle aux

méthodes Naming.bind() et Naming.lookup()

21.6.2. L'instanciation et l'enregistrement de l'objet distant

Il faut éxécuter la classe qui va instancier l'objet distant et l'enregistrer sous son nom dans le registre précédemment lancé.

Pour ne pas avoir de problème, il faut s'assurer que toutes les classes utiles (la classe de l'objet distant, l'interface qui définit les méthodes, le skeleton) sont présentes dans un répertoire défini dans la variable CLASSPATH.

21.6.3. Le lancement de l'application cliente

Cette section est en cours d'écriture

22. L'internationalisation

La localisation consiste à adapter un logiciel pour s'adapter aux caractéristiques locales de l'environnement d'execution telles que la langue. Le plus gros du travail consiste à traduire toutes les phrases et les mots. Les classes nécessaires sont incluses dans le packetage java.util.

Ce chapitre contient plusieurs sections :

- Les objets de type locale
- La classe ResourceBundle
- Chemins guidés pour réaliser la localisation
  Cette section propose plusieurs solutions pour réalisation l'internationalisation.

22.1. Les objets de type Locale

Un objet de type Locale identifie une langue et un pays donné.

22.1.1. Création d'un objet Locale

Exemple (code java 1.1) :

```
locale_US = new
Locale("en","US") ;

locale_FR = new Locale("fr","FR");
```

Le premier paramètre est le code langue (deux caractères minuscules conformes à la norme ISO−639 : exemple "de" pour l'allemand, "en" pour l'anglais, "fr" pour le français, etc ...)

Le deuxième paramètre est le code pays (deux caractères majuscules conformes à la norme ISO−3166 : exemple : "DE" pour l'Allemagne, "FR" pour la France,"US" pour les Etats Unis, etc ...). Ce paramètre est obligatoire : si le pays n'a pas besoin d'être précisé, il faut fournir une chaine vide.

Exemple (code java 1.1) :

```
Locale locale = new Locale("fr", "");
```

Un troisième paramètre peut permettre de préciser d'avantage la localisation par exemple la plateforme d'execution (il ne respecte aucun standard car il ne sera défini que dans l'application qui l'utilise) :

Exemple (code java 1.1) :

```
Locale locale_unix = new Locale("fr","FR", "UNIX");
Locale locale_windows = new Locale("fr","FR", "WINDOWS");
```

Ce troisième paramètre est optionnel.

La classe Locale définit des constantes pour certaines langues et pays :

Exemple (code java 1.1) : ces deux lignes sont equivalentes

```
Locale locale_1 = Locale.JAPAN;
Locale locale_2 = new Locale("ja", "JP");
```

Lorsque l'on précise une constante représentant une langue alors le code pays n'est pas défini.

Exemple (code java 1.1) : ces deux lignes sont equivalentes

```
Locale locale_3 = Locale.JAPANESE;
Locale locale_2 = new Locale("ja", "");
```

Il est possible de rendre la création d'un objet Locale dynamique :

Exemple (code java 1.1) :

```
static public void main(String[] args) {
   String langue = new String(args[0]);
   String pays = new String(args[1]);
   locale = new Locale(langue, pays);
}
```

Cet objet ne sert que d'identifiant qu'il faut passer à des objets de type ResourceBundle par exemple qui eux possèdent le nécessaire pour réaliser la localisation. En fait, le création d'un objet Locale pour un pays donné ne signifie pas que l'on va pouvoir l'utiliser.

22.1.2. Obtenir la liste des Locales disponibles

La méthode getAvailableLocales() permet de connaître la liste des Locales reconnues par une classe sensible à l'internationalisation

Exemple (code java 1.1) : avec la classe DateFormat

```
import java.util.*;
import java.text.*;

public class Available {
   static public void main(String[] args) {
      Locale liste[] =
      DateFormat.getAvailableLocales();
      for (int i = 0; i < liste.length; i++)
      {
         System.out.println(liste[i].toString());
         // toString retourne le code langue et le code pays séparé d'un souligné
      }
   }
}
```

La méthode Locale.getDisplayName() peut être utilisée à la place de toString pour obtenir le nom du code langue et du code pays.

22.1.3. L'utilisation d'un objet Locale

Il n'est pas obligatoire de se servir du même objet Locale avec les classes sensibles à l'internationnalisation.

Cependant la plupart des applications utilise l'objet Locale par défaut initialisé par la machine virtuelle avec les paramètres de la machine hote. La méthode Locale.getDefault() permet de connaître l'objet Locale par défaut.

22.2. La classe ResourceBundle

Il est préférable de définir un ResourceBundle pour chaque catégorie d'objet (exemple un par fénêtre) : ceci rend le code plus clair et plus facile à maintenir, évite d'avoir des ResourceBundle trop importants et réduit l'espace mémoire utilisé car chaque ressource n'est chargée que lorsque l'on en a besoin.

22.2.1. LA création d'un objet ResourceBundle

Conceptuellement, chaque ResourceBundle est un ensemble de sous classes qui partage la même racine de nom.

Exemple (code java 1.1) :

```
TitreBouton
TitreBouton_de
TitreBouton_en_GB
TitreBouton_fr_FR_UNIX
```

Pour sélectionner le ResourceBundle approprié il faut utiliser la méthode ResourceBundle.getBundle().

Exemple (code java 1.1) :

```
Locale locale = new Locale("fr", "FR");
ResourceBundle messages = ResourceBundle.getBundle("TitreBouton", locale);
```

Le premier argument contient le type d'objet à utiliser (la racine du nom de cet objet).

Le second argument de type Locale permet de déterminer quel fichier sera utilisé : il ajoute le code pays et le code langue séparé par un souligné à la racine du nom.

Si la classe désignée par l'objet Locale n'existe par, alors getBundle recherche celle qui se rapproche le plus. L'ordre de recherche sera le suivant :

Exemple (code java 1.1) :

```
TitreBouton_fr_CA_UNIX
TitreBouton_fr_FR
TitreBouton_fr
TitreBouton_en_US
TitreBouton_en
TitreBouton
```

Si aucune n'est trouvée alors getBundle lève une exception de type MissingResourceException.

22.2.2. Les sous classes de ResourceBundle : PropertyResourceBundle et ListResourceBundle

La classe abstraite ResourceBundle possède deux sous classes : PropertyResourceBundle et ListResourceBundle.

La classe ResourceBundle est une classe fléxible : le changement de l'utilisation d'un PropertyResourceBundle en ListResourceBundle se fait sans impact sur le code. La méthode getBundle() recherche le ResourceBundle désiré qu'il soit dans un fichier .class ou propriétés

22.2.2.1. L'utilisation de PropertyResourceBundle

Un PropertyResourceBundle est rattaché à un fichier propriétés. Ces fichiers propriétés ne font pas partie du code source java. Ils ne peuvent contenir que des chaines de caractères. Pour stocker d'autres objets, il faut utiliser des objets ListResourceBundle.

La création d'un fichier propriétés est simple : c'est un fichier texte qui contient des paires clé−valeur. La clé et la valeur sont séparées par un signe =. Chaque paire doit être sur une ligne séparée.

Exemple (code java 1.1) :

```
texte_suivant = suivant
texte_precedent = precedent
```

Le nom du fichier propriétés par défaut se compose de la racine du nom suivi de l'extension .properties.

Exemple : TitreBouton.properties.

Dans une autre langue, anglais par exemple, le fichier s'appelerait : TitreBouton_en.properties

Exemple (code java 1.1) :

```
texte_suivant = next
texte_precedent = previous
```

Les clés sont les mêmes, seule la traduction change.

Le nom de fichier TitreBouton_fr_FR.properties contient la racine (Titrebouton), le code langue (fr) et le code pays (FR).

22.2.2.2. L'utilisation de ListResourceBundle

La classe ListResourceBundle gère les ressources sous forme de liste encapsulée dans un objet. Chaque ListResourceBundle est donc rattaché à un fichier .class. On peut y stocker n'importe quel objet spécifique à la localisation.

Les objets ListResourceBundle contiennent des paires clé−valeur. La clé doit être une chaine qui caractérise l'objet. La valeur est un objet de n'importe quelle classe.

Exemple (code java 1.1) :

```
class TitreBouton_fr extends ListResourceBundle {
```

```
   public Object[][] getContents() {
      return contents;
   }
   static final Object[][] contents = {
      {"texte_suivant", "Suivant"},
      {"texte_precedent", "Precedent"},
   };
}
```

22.2.3. Obtenir un texte d'un objet ResourceBundle

La méthode getString() retourne la valeur de la clé précisée en paramètre.

Exemple (code java 1.1) :

```
String message_1 = messages.getString("texte_suivant");
String message_2 = TitreBouton.getString("texte_suivant");
```

22.3. Chemins guidés pour réaliser la localisation

22.3.1. L'utilisation d'un ResourceBundle avec un fichier propriétés

Il faut toujours créer le fichier propriété par défaut. Le nom de ce fichier commence avec le nom de base du ResourceBundle et se termine avec le suffix .properties

Exemple (code java 1.1) :

```
#Exemple de fichier propriété par défaut (TitreBouton.properties)
texte1 = suivant
texte2 = precedent
texte3 = quitter
```

Les lignes de commentaires commencent par un #. Les autres lignes contiennent les paires clé−valeur. Une fois le fichier défini, il ne faut plus modifier la valeur de la clé qui pourrait être appellée dans un programme.

Pour ajouter le support d'autre langue, il faut créer des fichier propriétés supplementaires qui contiendront les traductions. Le fichier est le même, seul le texte contenu dans la valeur change (la valeur de la clé doit être identique).

Exemple (code java 1.1) :

```
#Exemple de fichier propriété en anglais (TitreBouton_en.properties)
texte1 = next
texte2 = previous
texte3 = quit
```

Lors de la programmation, il faut créer un objet Locale. Il est possible de créer un tableau qui contient la liste des Locale disponibles en fonction des fichiers propriétés créés.

Exemple (code java 1.1) :

```
Locale[] locales = { Locale.GERMAN, Locale.ENGLISH };
```

Dans cet exemple, l'objet Locale.ENGLISH correspond au fichier TitreBouton_en.properties. L'objet Locale.GERMAN ne possédant pas de fichier propriétés défini, le fichier par défaut sera utilisé.

Il faut créer l'objet ResourceBundle en invoquant la méthode getBundle de l'objet Locale.

Exemple (code java 1.1) :

```
ResourceBundle titres =ResourceBundle.getBundle("TitreBouton", locales[1]);
```

La méthode getBundle() recherche en premier une classe qui correspond au nom de base, si elle n'existe pas alors elle recherche un fichier de propriétés. Lorsque le fichier est trouvé, elle retourne un objet PropertyResourceBundle qui contient les paires clé−valeur du fichier

Pour retrouver la traduction d'un texte, il faut utiliser la méthode getString() d'un objet ResourceBundle

Exemple (code java 1.1) :

```
String valeur = titres.getString(key);
```

Lors du deboggage, il peut être utile d'obtenir la liste de paire d'un objet ResourceBundle. La méthode getKeys() retourne un objet Enumeration qui contient toutes les clés de l'objet.

Exemple (code java 1.1) :

```
ResourceBundle titres =ResourceBundle.getBundle("TitreBouton", locales[1});
Enumeration cles = titres.getKeys();
while (bundleKeys.hasMoreElements()) {
   String cle = (String)cles.nextElement();
   String valeur = titres.getString(cle);
   System.out.println("cle = " + valeur +
      ", " +   "valeur = " + valeur);
}
```

22.3.2. Exemples de classes utilisant PropertiesResourceBundle

Exemple (code java 1.1) : Sources de la classe I18nProperties

```
/*
Test d'utilisation de la classe PropertiesResourceBundle
pour internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Description de la classe I18nProperties
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 */
public class I18nProperties {
  /**
   * Constructeur de la classe
   */
  public I18nProperties() {

    String texte;
    Locale locale;
    ResourceBundle res;

    System.out.println("Locale par defaut : ");
```

```
    locale = Locale.getDefault();
    res = ResourceBundle.getBundle("I18nPropertiesRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);

    System.out.println("\nLocale anglaise : ");
    locale = new Locale("en","");
    res = ResourceBundle.getBundle("I18nPropertiesRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);

    System.out.println("\nLocale allemande : "+
       "non définie donc utilisation locale par defaut ");
    locale = Locale.GERMAN;
    res = ResourceBundle.getBundle("I18nPropertiesRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);
  }

 /**
   * Pour tester la classe
   *
   * @param      args[]         arguments passes au programme
   */
  public static void main(String[] args) {
    I18nProperties i18nProperties = new I18nProperties();
  }

}
```

Exemple (code java 1.1) : Contenu du fichier I18nPropertiesRessources.properties

```
texte_suivant=suivant
texte_precedent=Precedent
```

Exemple (code java 1.1) : Contenu du fichier I18nPropertiesRessources_en.properties

```
texte_suivant=next
texte_precedent=previous
```

Exemple (code java 1.1) : Contenu du fichier I18nPropertiesRessources_en_US.properties

```
texte_suivant=next
texte_precedent=previous
```

22.3.3. L'utilisation de la classe ListResourceBundle

Il faut créer autant de sous classes de ListResourceBundle que de langues désirées : ceci va générer un fichier .class pour chacune des langues .

Exemple (code java 1.1) :

```
TitreBouton_fr_FR.class
TitreBouton_en_EN.class
```

Le nom de la classe doit contenir le nom de base plus le code langue et le code pays séparés par un souligné. A l'intérieur de la classe, un tableau à deux dimensions est initialisé avec les paires clé−valeur. Les clés sont des chaines qui doivent

être identiques dans toutes les classes des différentes langues. Les valeurs peuvent être des objets de n'importe quel type.

Exemple (code java 1.1) :

```
import java.util.*;

public class TitreBouton_fr_FR extends ListResourceBundle {
   public Object[][] getContents() {
      return contents;
   }

   private Object[][] contents = {
      { "texte_suivant", new String(" suivant ")},
      { "Numero", new Integer(4) }
   };
}
```

Il faut définir un objet de type Locale

Il faut créer un objet de type ResourceBundle en appelant la méthode getBundle() de la classe Locale

Exemple (code java 1.1) :

```
ResourceBundle titres=ResourceBundle.getBundle("TitreBouton", locale);
```

La méthode getBundle() recherche une classe qui commence par TitreBouton et qui est suivi par le code langue et le code pays précisé dans l'objet Locale passé en paramètre

La méthode getObject permet d'obtenir la valeur de la clé passée en paramètres. Dans ce cas une conversion est nécessaire.

Exemple (code java 1.1) :

```
Integer valeur = (Integer)titres.getObject("Numero");
```

22.3.4. Exemples de classes utilisant ListResourceBundle

Exemple (code java 1.1) : Sources de la classe I18nList

```
/*
Test d'utilisation de la classe ListResourceBundle
pour internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Description de la classe I18nList
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 */
public class I18nList {
  /**
   * Constructeur de la classe
   */
  public I18nList() {

    String texte;
    Locale locale;
    ResourceBundle res;
```

```
    System.out.println("Locale par defaut : ");
    locale = Locale.getDefault();
    res = ResourceBundle.getBundle("I18nListRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);

    System.out.println("\nLocale anglaise : ");
    locale = new Locale("en","");
    res = ResourceBundle.getBundle("I18nListRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);

    System.out.println("\nLocale allemande : "+
       "non définie donc utilisation locale par defaut ");
    locale = Locale.GERMAN;
    res = ResourceBundle.getBundle("I18nListRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);
  }

 /**
   * Pour tester la classe
   *
   * @param      args[]         arguments passes au programme
   */
  public static void main(String[] args) {
    I18nList i18nList = new I18nList();
  }

}
```

Exemple (code java 1.1) : Sources de la classe I18nListRessources

```
/*
test d'utilisation de la classe ListResourceBundle pour
internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Ressource contenant les traductions françaises
 * langue par defaut de l'application
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 *
 */

public class I18nListRessources extends ListResourceBundle {
  public Object[][] getContents() {
    return contents;
  }

   //tableau des mots clés et des valeurs

   static final Object[][] contents = {
   {"texte_suivant", "Suivant"},
   {"texte_precedent", "Precedent"},
  };
}
```

Exemple (code java 1.1) : Sources de la classe I18nListRessources_en

```
/*
test d'utilisation de la classe ListResourceBundle pour
internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Ressource contenant les traductions anglaises
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 *
 */
public class I18nListRessources_en extends ListResourceBundle {
  public Object[][] getContents() {
    return contents;
  }

   //tableau des mots clés et des valeurs

   static final Object[][] contents = {
   {"texte_suivant", "Next"},
   {"texte_precedent", "Previous"},
  };
}
```

Exemple (code java 1.1) : Sources de la classe I18nListRessources_en_US

```
/*
test d'utilisation de la classe ListResourceBundle pour
internationaliser une application
13/02/99
*/
import java.util.*;
/**
* Ressource contenant les traductions américaines
*
* @version 0.10 13 fevrier 1999
* @author Jean Michel DOUDOUX
*
*/

public class I18nListRessources_en_US extends ListResourceBundle {
public Object[][] getContents() {
return contents;
}

//tableau des mots clés et des valeurs

static final Object[][] contents = {
{"texte_suivant", "Next"},
{"texte_precedent", "Previous"},
};
}
```

22.3.5. La création de sa propre classe fille de ResourceBundle

La troisième solution consiste à créer sa propre sous classe de ResourceBundle et à surcharger la méthode handleGetObject().

Exemple (code java 1.1) :

```
abstract class MesRessources extends ResourceBundle {
```

```
   public Object handleGetObject(String cle)  {
      if(cle.equals(" texte_suivant "))
         return " Suivant " ;
      if(cle.equals(" texte_precedent "))
         return "Precedent " ;
         return null ;
   }
}
```

Attention : la classe ResourceBundle contient deux méthodes abstraites : handleGetObjects() et getKeys(). Si l'une des deux n'est pas définie alors il faut définir la sous classe avec le mot clé abstract.

Il faut créer autant de sous classes que de Locale désiré : il suffit simplement d'ajouter dans le nom de la classe le code langue et le code pays avec eventuellement le code variant.

22.3.6. Exemple de classes utilisant une classe fille de ResourceBundle

Exemple (code java 1.1) : Sources de la classe I18nResource

```
/*
Test d'utilisation d'un sous classement
de la classe ResourceBundle pour
internationaliser une application
13/02/99
*/
import java.util.*;
/**
 * Description de la classe I18nResource
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 */
public class I18nResource {
  /**
   * Constructeur de la classe
   */
  public I18nResource() {

    String texte;
    Locale locale;
    ResourceBundle res;


    System.out.println("Locale par defaut : ");
    res = ResourceBundle.getBundle("I18nResourceBundle");
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);


    System.out.println("\nLocale anglaise : ");
    locale = new Locale("en","");
    res = ResourceBundle.getBundle("I18nResourceBundle", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);

    System.out.println("\nLocale allemande : "+
       "non définie donc utilisation locale par defaut ");
    locale = Locale.GERMAN;
    res = ResourceBundle.getBundle("I18nResourceBundle", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
```

```
    System.out.println("texte_precedent = "+texte);

  }

 /**
   * Pour tester la classe
   *
   * @param      args[]         arguments passes au programme
   */
  public static void main(String[] args) {
    I18nResource i18nResource = new I18nResource();
  }

}
```

Exemple (code java 1.1) : Sources de la classe I18nResourceBundle

```
/*
Test d'utilisation de la derivation de la classe ResourceBundle pour
internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Description de la classe I18nResourceBundle
 * C'est la classe contenant la locale par defaut
 * Contient les traductions de la locale francaise (langue par defaut)
 * Elle herite de ResourceBundle
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 */
public class I18nResourceBundle extends ResourceBundle {
   protected Vector table;

   public I18nResourceBundle() {
      super();
      table = new Vector();
      table.addElement("texte_suivant");
      table.addElement("texte_precedent");
   }

   public Object handleGetObject(String cle)  {
      if(cle.equals(table.elementAt(0))) return "Suivant" ;
      if(cle.equals(table.elementAt(1))) return "Precedent" ;
      return null ;
   }

   public Enumeration getKeys()  {
      return table.elements();
   }

}
```

Exemple (code java 1.1) : Sources de la classe I18nResourceBundle_en

```
/*
Test d'utilisation de la derivation de la classe ResourceBundle pour
internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Description de la classe I18nResourceBundle_en
 * Contient les traductions de la locale anglaise
```

```
 * Elle herite de la classe contenant la locale par defaut
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 */
public class I18nResourceBundle_en extends I18nResourceBundle {
   public Object handleGetObject(String cle)  {
      if(cle.equals(table.elementAt(0))) return "Next" ;
      if(cle.equals(table.elementAt(1))) return "Previous" ;
      return null ;
   }
}
```

Exemple (code java 1.1) : Sources de la classe I18nResourceBundle_fr_FR

```
/*
Test d'utilisation de la derivation de la classe ResourceBundle pour
internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Description de la classe I18nResourceBundle_fr_FR
 * Contient les traductions de la locale francaise
 * Elle herite de la classe contenant la locale par defaut
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 */
public class I18nResourceBundle_fr_FR extends I18nResourceBundle {

   /**
    *
    * Retourne toujours null car la locale francaise correspond
    * a la locale par defaut
    *
    */
   public Object handleGetObject(String cle)  {
      return null ;
   }
}
```

23. Les composants java beans

Les java beans sont des composants réutilisables introduits par le JDK 1.1. De nombreuses fonctionnalités de ce JDK lui ont été ajoutées pour développer des caractéristiques de ces composants. Les java beans sont couramment appelés simplement beans.

Les beans sont prévues pour pouvoir inter−agir avec d'autres beans au point de pouvoir développer une application simplement en assemblant des beans avec un outil graphique dédié. Sun fournit gratuitement un tel outils : le B.D.K. (Bean Development Kit).

23.1. Présentations des java beans

Des composants réutilisables sont des objets autonomes qui doivent pouvoir être facilement assemblés entres eux pour créer un programme.

Microsoft propose la technologie ActiveX pour définir des composants mais celle ci est spécifiquement destinée aux plate−formes Windows.

Les java beans proposé par Sun reposent bien sûre sur java et de fait en possède toutes les caractéristiques : indépendance de la plate−forme, taille réduite du composant, ...

La technologie java beans propose de simplifier et faciliter la création et l'utilisation de composants.

Les java beans possèdent plusieurs caractéristiques :

- la persistance : elle permet grâce au mécanisme de sérialisation de sauvegarder l'état d'un bean pour le restaurer ainsi si on assemble plusieurs beans pour former une application, on peut la sauvegarder.
- la communication grâce à des événements qui utilise le modèle des écouteurs introduit par java 1.1
- l'introspection : ce mécanisme permet de découvrir de façon dynamique l'ensemble des éléments qui compose le bean (attributs, méthodes et événements) sans avoir le source.
- la possibilité de paramétrer le composant : les données du paramétrage sont conservées dans des propriétés.

Ainsi, les beans sont des classes java qui doivent respecter un certains nombre de règles :

- ils doivent posséder un constructeur sans paramètre. Celui ci devra initialiser l'état du bean avec des valeurs par défauts.
- ils peuvent définir des propriétés : celles ci sont identifiées par des méthodes dont le nom et la signature sont normalisés
- ils devraient implémenter l'interface serialisable : ceci est obligatoire pour les beans qui possèdent une partie graphique pour permettre la sauvegarde de leur état
- ils définissent des méthodes utilisables par les composants extérieures : elles doivent être public et prévoir une gestion des accès concurrents
- ils peuvent émettent des événements en gérant une liste d'écouteurs qui s'y abonnent via des méthodes dont les noms sont normalisés

Le type de composants le plus adapté est le composant visuel. D'ailleurs, les composants des classes A.W.T. et Swing pour la création d'interfaces graphiques sont tous des beans. Mais les beans peuvent aussi être des composants non visuels pour prendre en charge les traitements.

23.2. Les propriétés.

Les propriétés contiennent des données qui gèrent l'état du composant : ils peuvent être de type primitif ou être un objet.

Il existe quatres types de propriétés :

- les propriétés simples
- les propriétés indexées (indexed properties)
- les propriétés liées (bound properties)
- les propriétés liées avec contraintes (Constrained properties)

23.2.1. Les propriétés simples

Les propriétés sont des variables d'instance du bean qui possèdent des méthodes particulières pour lire et modifier leur valeur. La normalisation de ces méthodes permet à des outils de déterminer de façon dynamique quels sont les propriétés du bean. L'accès à ces propriétés doit se faire via ces méthodes. Ainsi la variable qui stocke la valeur de la une propriété ne doit pas être déclarée public mais les méthodes d'accès doivent bien sûre l'être.

Le nom de la méthode de lecture d'une propriété doit obligatoirement commencer par « get » suivi par le nom de la propriété dont la première lettre doit être une majuscule. Une telle méthode est souvent appelée « getter » ou « accesseur » de la propriété. La valeur retournée par cette méthode doit être du type de la propriété.

Exemple (code java 1.1) :

```
private int longueur;
public int getLongueur () {
  return longueur;
}
```

Pour les propriétés booléennes, une autre convention peut être utilisée : la méthode peut commencer par «is» au lieu de « get ». Dans ce cas, la valeur de retour est obligatoirement de type boolean.

Le nom de la méthode permettant la modification d'une propriété doit obligatoirement commencer par « set » suivi par le nom de la propriété dont la première lettre doit être une majuscule. Une telle méthode est souvent appelée « setter ». Elle ne retourne aucune valeur et doit avoir en paramètre une variable du type de la propriété qui contiendra sa nouvelle valeur. Elle devra assurer la mise à jour de la valeur de la propriété en effectuant éventuellement des contrôles et/ou des traitements (par exemple le rafraîchissement pour un bean visuel dont la propriété affecte l'affichage).

Exemple (code java 1.1) :

```
private int longueur ;
public void setLongueur (int longueur) {
  this.longueur = longueur;
}
```

Une propriété peut n'avoir qu'un getter et pas de setter : dans ce cas, la propriété n'est utilisable qu'en lecture seule.

Le nom de la variable d'instance qui contient la valeur de la propriété n'est pas obligatoirement le même que le nom de la propriété

Il est préférable d'assurer une gestion des accès concurrents dans ces méthodes de lecture et de mise à jour des propriétés par exemple en déclarant ces méthodes synchronized.

Les méthodes du beans peuvent directement manipuler en lecture et écriture la variable d'instance qui stocke la valeur de la propriété, mais il est préférable d'utiliser le getter et le setter.

23.2.2. les propriétés indexées (indexed properties)

Ce sont des propriétés qui possèdent plusieurs valeurs stockées dans un tableau.

Pour ces propriétés, il faut aussi définir des méthodes « get » et « set » dont il convient d'ajouter un paramètre de type int représentant l'index de l'élément du tableau.

Exemple (code java 1.1) :

```
private float[] notes = new float[5];
public float getNotes (int i ) {
  return notes[i];
}
public void setNotes (int i ; float notes) {
  this.notes[i] = notes;
}
```

Il est aussi possible de définir des méthodes « get » et « set » permettant de mettre à jour tout le tableau.

Exemple (code java 1.1) :

```
private float[] notes = new float[5] ;
public float[] getNotes () {
  return notes;
}
public void setNotes (float[] notes) {
  this.notes = notes;
}
```

23.2.3. Les propriétés liées (Bound properties)

Il est possible d'informer d'autre composant du changement de la valeur d'une propriété d'un bean. Les java beans peuvent mettre en place un mécanisme qui permet pour une propriété d'enregistrer des composants qui seront informés du changement de la valeur de la propriété.

Ce mécanisme peut être mis en place grâce à un objet de la classe PropertyChangeSupport qui permet de simplifier la gestion de la liste des écouteurs et de les informer des changements de valeur d'une propriété. Cette classe définie les méthodes addPropertyChangeListener() pour enregistrer un composant désirant être informé du changement de la valeur de la propriété et removePropertyChangeListener() pour supprimer un composant de la liste.

La méthode firePropertyChange() permet d'informer tous les composants enregistrés du changement de la valeur de la propriété.

Le plus simple est que le bean hérite de cette classe si possible car les méthodes addPropertyChangeListener() et removePropertyChangeListener() seront directement héritées.

Si ce n'est pas possible, il est obligatoire de définir les méthodes addPropertyChangeListener() et removePropertyChangeListener() dans le bean qui appelleront les méthodes correspondantes de l'objet PropertyChangeSupport.

Exemple (code java 1.1) :

```
import java.io.Serializable;
import java.beans.*;
public class MonBean03 implements Serializable {
  protected int valeur;

  PropertyChangeSupport changeSupport;

  public MonBean03(){
    valeur = 0;

    changeSupport = new PropertyChangeSupport(this);
  }

  public synchronized void setValeur(int val) {
    int oldValeur = valeur;
    valeur = val;

    changeSupport.firePropertyChange("valeur",oldValeur,valeur);
  }
  public synchronized int getValeur() {
    return valeur;
  }
  public synchronized void addPropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.addPropertyChangeListener(listener);
  }

  public synchronized void removePropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.removePropertyChangeListener(listener);
  }

}
```

Les composants qui désirent être enregistrés doivent obligatoirement implémenter l'interface PropertyChangeListener et définir la méthode propertyChange() déclarée par cette interface.

La méthode propertyChange() reçoit en paramètre un objet de type PropertyChangeEvent qui représente l'événement. Cette méthode de tous les objets enregistrés est appelée. Le paramètre de type PropertyChangeEvent contient plusieurs informations :

- l'objet source : le bean dont la valeur d'une propriété a changé
- le nom de la propriété sous forme de chaîne de caractères
- l'ancienne valeur sous forme d'un objet de type Object
- la nouvelle valeur sous forme d'un objet de type Object

Pour les traitements, il est souvent nécessaire d'utiliser un cast pour transmettre ou utiliser les objets qui représentent l'ancienne et la nouvelle valeur.

Méthode	Rôle
public Object getSource()	retourne l'objet source
public Object getNewValue()	retourne la nouvelle valeur de la propriété
public Object getOldValue()	retourne l'ancienne valeur de la propriété
public String getPropertyName	retourne le nom de la propriété modifiée

Exemple (code java 1.1) : un programme qui créé le bean et lui associe un écouteur

```
import java.beans.*;
import java.util.*;
public class TestMonBean03 {
  public static void main(String[] args) {
    new TestMonBean03();
  }
```

```
  public TestMonBean03() {
    MonBean03 monBean = new MonBean03();

    monBean.addPropertyChangeListener( new PropertyChangeListener() {
      public void propertyChange(PropertyChangeEvent event) {
        System.out.println("propertyChange : valeur = "+ event.getNewValue());
      }
    } );

    System.out.println("valeur = " + monBean.getValeur());
    monBean.setValeur(10);
    System.out.println("valeur = " + monBean.getValeur());

  }
}
```

Résultat :

```
C:\tutorial\sources exemples>java TestMonBean03
valeur = 0
propertyChange : valeur = 10
valeur = 10
```

Pour supprimer un écouteur de la liste du bean, il suffit d'appeler la méthode removePropertyChangeListener() en lui passant en paramètre un référence sur l'écouteur.

23.2.4. Les propriétés liées avec contraintes (Constrained properties)

Ces propriétés permettent à un ou plusieurs composants de mettre un veto sur la modification de la valeur de la propriété.

Comme pour les propriétés liées, le bean doit gérer un liste de composants « écouteurs » qui souhaitent être informé d'un changement possible de la valeur de la propriété. Si un composant désire s'opposer à ce changement de valeur, il lève une exception pour en informer le bean.

Les écouteurs doivent implémenter l'interface VetoableChangeListener qui définie la méthode vetoableChange().

Avant le changement de la valeur, le bean appelle cette méthode vetoableChange() de tous les écouteurs enregistrés. Elle possède en paramètre un objet de type PropertyChangeEvent qui contient : le bean, le nom de la propriété, l'ancienne valeur et la nouvelle valeur.

Si un écouteur veut s'opposer à la mise à jour de la valeur, il lève une exception de type java.beans.PropertyVetoException. Dans ce cas, le bean ne change pas la valeur de la propriété : ces traitements sont à la charge du programmeur avec notamment la gestion de la capture et du traitement de l'exception dans un bloc try/catch.

La classe VetoableChangeSupport permet de simplifier la gestion de la liste des écouteurs et de les informer du futur changement de valeur d'une propriété. Son utilisation est similaire à celle de la classe PropertyChangeSupport.

Pour ces propriétés, pour que les traitements soient complets il faut implémenter le code pour gérer et traiter les écouteurs qui souhaitent connaître les changements de valeur effectifs de la propriété (voir les propriétés liées).

Exemple (code java 1.1) :

```
import java.io.Serializable;
import java.beans.*;
public class MonBean04 implements Serializable {
  protected int oldValeur;
  protected int valeur;

  PropertyChangeSupport changeSupport;
  VetoableChangeSupport vetoableSupport;
```

```
  public MonBean04(){
    valeur = 0;
    oldValeur = 0;

    changeSupport = new PropertyChangeSupport(this);
    vetoableSupport = new VetoableChangeSupport(this);
  }


  public synchronized void setValeur(int val) {
    oldValeur = valeur;
    valeur = val;

    try {
      vetoableSupport.fireVetoableChange("valeur",new Integer(oldValeur),new Integer(valeur));
    } catch(PropertyVetoException e) {
      System.out.println("MonBean, un veto est emis :  "+e.getMessage());
      valeur = oldValeur;
    }
    if ( valeur != oldValeur ) {
      changeSupport.firePropertyChange("valeur",oldValeur,valeur);
    }

  }
  public synchronized int getValeur() {
    return valeur;
  }
  public synchronized void addPropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.addPropertyChangeListener(listener);
  }

  public synchronized void removePropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.removePropertyChangeListener(listener);
  }
  public synchronized void addVetoableChangeListener(VetoableChangeListener listener) {
    vetoableSupport.addVetoableChangeListener(listener);
  }
  public synchronized void removeVetoableChangeListener(VetoableChangeListener listener) {
    vetoableSupport.removeVetoableChangeListener(listener);
  }

}
```

Exemple (code java 1.1) : un programme qui teste le bean. Il émet un veto si la nouvelle valeur de la propriété est supérieure à 100.

```
import java.beans.*;
import java.util.*;
public class TestMonBean04 {
  public static void main(String[] args) {
    new TestMonBean04();
  }

  public TestMonBean04() {
    MonBean04 monBean = new MonBean04();

    monBean.addPropertyChangeListener( new PropertyChangeListener() {
      public void propertyChange(PropertyChangeEvent event) {
        System.out.println("propertyChange : valeur = "+ event.getNewValue());
      }
    } );

    monBean.addVetoableChangeListener( new VetoableChangeListener() {

      public void vetoableChange(PropertyChangeEvent event) throws PropertyVetoException {
        System.out.println("vetoableChange : valeur = " + event.getNewValue());
        if( ((Integer)event.getNewValue()).intValue() > 100 )
          throw new PropertyVetoException("valeur superieur a 100",event);
      }
    } );
```

```
    System.out.println("valeur = " + monBean.getValeur());
    monBean.setValeur(10);
    System.out.println("valeur = " + monBean.getValeur());
    monBean.setValeur(200);
    System.out.println("valeur = " + monBean.getValeur());

  }
}
```

Résultat :

```
C:\tutorial\sources exemples>java TestMonBean04
valeur = 0
vetoableChange : valeur = 10
propertyChange : valeur = 10
valeur = 10
vetoableChange : valeur = 200
vetoableChange : valeur = 10
MonBean, un veto est emis :  valeur superieur a 100
valeur = 10
```

23.3. Les méthodes

Toutes les méthodes publiques sont visibles de l'extérieures et peuvent donc être appelées.

23.4. Les événements

Les beans utilisent les événements définis dans le modèle par délégation introduit par le J.D.K. 1.1 pour dialoguer. Par respect de ce modèle, le bean est la source et les autres composants qui souhaitent être informé sont nommé « Listeners » ou « écouteurs » et doivent s'enregistrer auprès du bean qui maintient la liste des composants enregistrés.

Il est nécessaire de définir les méthodes qui vont permettre de gérer la liste des écouteurs désirant recevoir l'événement. Il faut définir deux méthodes :

- public void addXXXListener(XXXListener li) pour enregistrer l'écouteur li
- public void removeXXXListener (XXXListener li) pour enlever l'écouteur li de la liste

L'objet de type XXXListener doit obligatoirement implémenter l'interface java.util.EventListener et son nom doit terminer par « Listener ».

Les événements peuvent être mono ou multi écouteurs.

Pour les évenements mono écouteurs, la méthode addXXXListener() doit indiquer dans sa signature qu'elle est succeptible de lever l'exception java.util.TooManyListenersException si un écouteurs tente de s'enregistrer et qu'il y en a déjà un présent.

23.5. L'introspection

L'introspection est un mécanisme qui permet de déterminer de façon dynamique les caractéristiques d'une classe et donc d'un bean. Les caractéristiques les plus importantes sont les propriétés, les méthodes et les événements. Le principe de l'introspection permet à Sun d'éviter de rajouter des éléments au langage pour définir ces caractéristiques.

L'API JavaBean définie la classe java.beans.Introspector qui facilite et standardise la recherche des propriétés, méthodes et événements du bean. Cette classe possède des méthodes pour analyser le bean et retourner un objet de type BeanInfo contenant les informations trouvées.

La classe Introspector utilise deux techniques pour retrouver ces informations :

1. un objet de type BeanInfo, si il y en a un défini par les développeurs du bean
2. les mécanismes fournis par l'API reflexion pour extraire les entités qui respectent leur modèle (design pattern) respectif.

Il est donc possible de définir un objet BeanInfo qui sera directement utilisé par la classe Introspector. Cette définition est utile si le bean ne respecte pas certains modèles (designs patterns) ou si certaines entités héritées ne doivent pas être utilisables. Dans ce cas, le nom de cette classe doit obligatoirement respecter le modèle XXXBeanInfo ou XXX est le nom du bean correspondant. La classe Introspector recherche une classe respectant ce modèle.

Si une classe BeanInfo pour un bean est définie, une classe qui hérite du bean n'est pas obligée de définir un classe BeanInfo. Dans ce cas, la classe Introspector utilise les informations du BeanInfo de la classe mère et ajoute les informations retournée par l'API reflection sur le bean.

Sans classe BeanInfo associée au bean, les méthodes de la classe Introspector utilisent les techniques d'introspection pour analyser le bean.

23.5.1. Les modèles (designs patterns)

Si la classe Introspector utilise l'API reflection pour déterminer les informations sur le bean et utilise en même temps un ensembles de modèles sur chacunes des entités propriétés, méthodes et événements.

Pour déterminer les propriétés, la classe Introspector recherche les méthodes getXxx, setXxx et isXxx ou Xxx représente le nom de la propriété dont la première lettre est en majuscule. La première lettre du nom de la propriété est remise en minuscule sauf si les deux premières lettres de la propriété sont en majuscules.

Pour déterminer les méthodes, la classe Introspector rechercher toutes les méthodes publiques.

Pour déterminer les événements, la classe Introspector recherche les méthodes addXxxListener() et removeXxxListener(). Si les deux sont présentes, elle en déduit que l'événement xxx est défini dans le bean. Comme pour les propriétés, la première lettre du nom de l'événement est mis en minuscule.

23.5.2. La classe BeanInfo

La classe BeanInfo contient des informations sur un bean et possède plusieurs méthodes pour les obtenir.

La méthode getBeanInfo() prend en paramètre un objet de type Class qui représente la classe du bean et elle renvoie des informations sur la classe et toutes ses classes mères.

Une version surchargée de la méthode accepte deux objets de type Class : le premier représente le bean et le deuxième représente une classe appartenant à la hiérarchie du bean. Dans ce cas, la recherche d'informations d'arrêtera juste avant d'arriver à la classe précisée en deuxième argument.

Exemple : obtenir des informations sur le bean uniquement (sans informations sur ces super classes)

Exemple (code java 1.1) :

```
Class monBeanClasse = Class.forName("monBean");
BeanInfo bi = Introspector.getBeanInfo(monBeanClasse, monBeanClasse.getSuperclass());
```

La méthode getBeanDescriptor() permet d'obtenir des informations générales sur le bean en renvoyant un objet de type BeanDescriptor()

La méthode getPropertyDescriptors() permet d'obtenir un tableau d'objets de type PropertyDescriptor qui contient les caractéristiques d'une propriété. Plusieurs méthodes permettent d'obtenir ces informations.

Exemple (code java 1.1) :

```
    PropertyDescriptor[] propertyDescriptor = bi.getPropertyDescriptors();
    for (int i=0; i<propertyDescriptor.length; i++) {
      System.out.println(" Nom propriete    : " +
                         propertyDescriptor[i].getName());
      System.out.println(" Type propriete   : "
                + propertyDescriptor[i].getPropertyType());
      System.out.println(" Getter propriete : "
                 + propertyDescriptor[i].getReadMethod());
      System.out.println(" Setter propriete : "
                + propertyDescriptor[i].getWriteMethod());
    }
```

La méthode getMethodDescriptors() permet d'obtenir un tableau d'objets de type MethodDescriptor qui contient les caractéristiques d'une méthode. Plusieurs méthodes permettent d'obtenir ces informations.

Exemple (code java 1.1) :

```
    MethodDescriptor[] methodDescriptor;
    unMethodDescriptor = bi.getMethodDescriptors();
    for (int i=0; i < unMethodDescriptor.length; i++) {
      System.out.println(" Methode : "+unMethodDescriptor[i].getName());
    }
```

La méthode getEventSetDescriptors() permet d'obtenir un tableau d'objets de type EventSetDescriptor qui contient les caractéristiques d'un événement. Plusieurs méthodes permettent d'obtenir ces informations.

Exemple (code java 1.1) :

```
    EventSetDescriptor[] unEventSetDescriptor = bi.getEventSetDescriptors();
    for (int i = 0; i < unEventSetDescriptor.length; i++) {
      System.out.println(" Nom evt             : "
                       + unEventSetDescriptor[i].getName());
      System.out.println(" Methode add evt     : " +
             unEventSetDescriptor[i].getAddListenerMethod());
      System.out.println(" Methode remove evt  : " +
          unEventSetDescriptor[i].getRemoveListenerMethod());
      unMethodDescriptor = unEventSetDescriptor[i].getListenerMethodDescriptors();
      for (int j = 0; j < unMethodDescriptor.length; j++) {
          System.out.println(" Event Type: " + unMethodDescriptor[j].getName());
      }
    }
```

Exemple complet (code java 1.1) :

```
import java.io.*;
import java.beans.*;
import java.lang.reflect.*;
public class BeanIntrospection {
  static String nomBean;
  public static void main(String args[]) throws Exception {
    nomBean = args[0];
    new BeanIntrospection();
  }

  public BeanIntrospection() throws Exception {
    Class monBeanClasse = Class.forName(nomBean);
    MethodDescriptor[] unMethodDescriptor;

    BeanInfo bi = Introspector.getBeanInfo(monBeanClasse, monBeanClasse.getSuperclass());
    BeanDescriptor unBeanDescriptor = bi.getBeanDescriptor();
    System.out.println("Nom du bean    : " + unBeanDescriptor.getName());
    System.out.println("Classe du bean : " + unBeanDescriptor.getBeanClass());
    System.out.println("");
```

```
    PropertyDescriptor[] propertyDescriptor = bi.getPropertyDescriptors();
    for (int i=0; i<propertyDescriptor.length; i++) {
      System.out.println(" Nom propriete    : " +
                         propertyDescriptor[i].getName());
      System.out.println(" Type propriete   : "
                + propertyDescriptor[i].getPropertyType());
      System.out.println(" Getter propriete : "
                 + propertyDescriptor[i].getReadMethod());
      System.out.println(" Setter propriete : "
                 + propertyDescriptor[i].getWriteMethod());
    }
    System.out.println("");
    unMethodDescriptor = bi.getMethodDescriptors();
    for (int i=0; i < unMethodDescriptor.length; i++) {
      System.out.println(" Methode : "+unMethodDescriptor[i].getName());
    }
    System.out.println("");
    EventSetDescriptor[] unEventSetDescriptor = bi.getEventSetDescriptors();
    for (int i = 0; i < unEventSetDescriptor.length; i++) {
      System.out.println(" Nom evt             : "
                       + unEventSetDescriptor[i].getName());
      System.out.println(" Methode add evt     : " +
             unEventSetDescriptor[i].getAddListenerMethod());
      System.out.println(" Methode remove evt  : " +
          unEventSetDescriptor[i].getRemoveListenerMethod());
      unMethodDescriptor = unEventSetDescriptor[i].getListenerMethodDescriptors();
      for (int j = 0; j < unMethodDescriptor.length; j++) {
          System.out.println(" Event Type: " + unMethodDescriptor[j].getName());
      }
    }
    System.out.println("");
  }
}
```

23.6. Paramétrage du bean (Customization)

Il est possible de développer un éditeur de propriétés spécifique pour permettre de personnaliser la modification des paramètres du bean.

Cette section est en cours d'écriture

23.7. La persistance

Les propriétés du bean doivent pouvoir être sauvés pour être restitués ultérieurement. Le mécanisme utilisé est la sérialisation. Pour permettre d'utiliser ce mécanisme, le bean doit implémenter l'interface Serializable

23.8. La diffusion sous forme de jar

Pour diffuser un bean sous forme de jar, il faut définir un fichier manifest.

Ce fichier doit obligatoirement contenir un attribut Name: qui contient le nom complet de la classe (incluant le package) et un attribut Java−Bean: valorisé à True.

Exemple de fichier manifest pour un bean :

```
Name: MonBean.class
Java-Bean: True
```

Cette section est en cours d'écriture

23.9. Le B.D.K.

L'outils principal du B.D.K. est la BeanBox. Cet outils écrit en java permet d'assembler des beans.

Pour utiliser un bean dans la BeanBox, il faut qu'il soit utilisé sous forme d'archive jar.

Cette section est en cours d'écriture

24. Logging

Le logging est important dans toutes les applications pour permettre de faciliter le debuggage lors du développement et de conserver une trace de son exécution lors de l'exploitation en production.

Une API très répandue est celle développée par le projet open source log4j développé par le groupe Jakarta.

Concient de l'importance du logging, Sun a développé et intégré au JDK 1.4 une API dédiée. Cette API est plus simple à utiliser et elle offre moins de fonctionnalités que log4j mais elle a l'avantage d'être fournie directement avec le JDK.

Ce chapitre est en cours d'écriture

24.1. Log4j

Log4j est un projet open source distribué sous la licence Apache Software. La dernière version de log4j est téléchargeable à l'url http://jakarta.apache.org/log4j/ : elle inclus les binaires (les fichiers .class), les sources complètes et la documentation. Log4j est compatible avec le JDK 1.1.

Cette API permet aux développeurs d'utiliser et de paramétrer les traitements de logs. Il est possible de fournir les paramètres de l'outils dans un fichier de configuration. Log4j est thread−safe.

Log4j utilise trois composants principaux :

- Categories : ils premettent de gérer les logs
- Appenders : ils représentent les flux qui vont recevoir les messages de log
- Layouts : ils permettent de formatter le contenu des messages de la log

24.1.1. Les catégories

Les catégories déterminent si un message doit être envoyé dans la ou les logs. Elles sont représentées par la classe org.apache.log4j.Category

Chaque catégorie possède un nom qui est sensible à la casse. Pour créer une categorie, il suffit d'instancier un objet Category. Pour réaliser cette instanciation, la classe Category fournie une méthode statique getInstance() qui attend en paramètre le nom de la categorie. Si une catégorie existe déjà avec le nom fournie, alors la méthode getInstance() renvoie une instance sur cette catégorie.

Il est pratique de fournir le nom complet de la classe comme nom de la catégorie dans laquelle elle est instanciée mais ce n'est pas obligation. Il est ainsi possible de créer une hiérarchie spécifique différente de celle de l'application, par exemple basée sur des aspects fonctionnels. L'inconvénient d'associer le nom de la classe au nom de la catégorie est qu'il faut instancier un objet Category dans chaque classe. Le plus pratique est de déclarer cette objet static.

Exemple :

```
public class Classe1 {
  static Category category = Category.getInstance(Classe1.class.getName());
  ...
}
```

La méthode log(Priority, Object) permet de demander l'envoie dans la log du message avec la priorité fournie.

Il existe en plus une méthode qui permet de demander l'envoie d'un message dans la log pour chaque priorité : debug(Object), info(Object), warn(Object), error(Object), fatal(Object).

La demande est traitée en fonction de la hiérarchie de la catégorie et de la priorité du message et de cette de la catégorie.

Pour éviter d'éventuels traitements pour créer le message, il est possible d'utiliser la méthode isEnabledFor(Priority) pour savoir si la catégorie traite la priorité ou non

Exemple :

```
import org.apache.log4j.*;

public class TestIsEnabledFor {

  static Category cat1 = Category.getInstance(TestIsEnabledFor.class.getName());

  public static void main(String[] args) {
    int i=1;
    int[] occurence={10,20,30};

    BasicConfigurator.configure();

    cat1.setPriority(Priority.WARN) ;
    cat1.warn("message de test");

    if(cat1.isEnabledFor(Priority.INFO)) {
      System.out.println("traitement du message de priorité INFO");
      cat1.info("La valeur de l'occurence "+i+" = " + String.valueOf(occurence[i]));
    }
    if(cat1.isEnabledFor(Priority.WARN)) {
      System.out.println("traitement du message de priorité WARN");
      cat1.warn("La valeur de l'occurence "+i+" = " + String.valueOf(occurence[i]));
    }

  }
}
```

Résultat :

```
0 [main] WARN TestIsEnabledFor  − message de test
traitement du message de priorit_ WARN
50 [main] WARN TestIsEnabledFor  − La valeur de l'occurence 1 = 20
```

24.1.1.1. La hiérarchie dans les catégories

Le nom de la categorie permet de la placer dans une hierarchie dont la racine est une catégorie spéciale nommée root qui est créée par défaut sans nom.

La classe Category possède une classe statique getRoot() pour obtenir la catégorie racine.

La hiérarchie des noms est établie grace à la notation par point comme pour les packages.

Exemple : soit trois catégories
categorie1 nommée "org"
categorie2 nommée "org.moi"
categorie3 nommée "org.moi.projet"

Categorie3 est fille de categorie2, elle même fille de categorie1.

Cette relation hiérarchique est importante car la configuration établie pour une catagorie est automatiquement propagée par défaut aux categories enfants.

L'ordre de la création des catégories de la hiérarchie ne doit pas obligatoirement respecter l'ordre de la hiérarchie. Celle ci est constituée au fur et à mesure de la création des catégories.

24.1.1.2. Les priorités

Log4j gére des priorités pour permettre à la categorie de déterminer si le message sera envoyé dans la log. Il existe cinq priorités qui possèdent un ordre hiérarchique croissant :

- DEBUG
- INFO
- WARN
- ERROR
- FATAL

La classe org.apache.log4j.Priority encapsule ces priorités.

Chaque catégorie est associé à une priorité qui peut être changée dynamiquement. La catégorie détermine si un message doit être envoyé dans la log en comparant sa priorité avec la priorité du message. Si celle est supérieure ou égale à la priorité de la catégorie, alors le message est envoyé dans la log.

La méthode setPropriety() de la classe Category permet de préciser la priorité de la categorie.

Si aucune priorité n'est donnée à une catégorie, elle "hérite" de la priorité de la première catégorie en remontant dans la hiérarchie dont la priorité est renseignée.

Exemple : soit trois catégories
root associée à la priorité INFO
categorie1 nommée "org" sans priorité particulière
categorie2 nommée "org.moi" associée à la priorité ERROR
categorie3 nommée "org.moi.projet" sans priorité particulière

Une demande avec la priorité DEBUG sur categorie2 n'est pas traitée car la priorité INFO héritée est supérieure à DEBUG.
Une demande avec la priorité WARN sur categorie2 est traitée car la priorité INFO héritée est inférieure à WARN .
Une demande avec la priorité DEBUG sur categorie3 n'est pas traitée car la priorité ERROR héritée est supérieure à DEBUG.
Une demande avec la priorité FATAL sur categorie3 est traitée car la priorité ERROR héritée est inférieure à FATAL.
En fait dans l'exemple, aucune demande avec la priorité DEBUG ne sera traitée.

Au niveau applicatif, il est possible d'interdire le traitement d'une priorité et de celle inférieure en utilisant le code suivant : Category.getDefaultHierarchy().disable(). Il faut fournir la priorité à la méthode disable().

Il est possible d'annuler ce traitement dynamiquement en positionnement la propriété système : log4j.disableOverride.

24.1.2. Les Appenders

L'interface org.apache.log4j.Appender désigne un flux qui représente la log et se charge de l'envoie de message formatté ce flux. Le formattage proprement dit est réalisé par un objet de type Layout. Ce layout peut être fourni dans le constructeur adapté ou par la méthode setLayout().

Une catégorie peut posséder plusieurs appenders. Si la catégorie décide de traiter la demande de message, le message est envoyés à chacun des appenders. Pour ajouter un appender à un catégorie, il suffit d'utiliser la méthode addAppender() qui attend en paramètre un objet de type Appender.

L'interface Appender est directement implémentée par la classe abstraite AppenderSkeleton. Cette classe est la classe mère de toutes les classes fournies avec log4j pour représenter un type de log :

- AsyncAppender
- JMSAppender
- NTEventLogAppender : log envoyée dans la log des évenements sur un système Windows NT
- NullAppender
- SMTPAppender
- SocketAppender : log envoyées dans une socket
- SyslogAppender : log envoyée dans le demon syslog d'un système Unix
- WriterAppender : cette classe possède deux classes filles : ConsoleAppender et FileAppender. La classe FileAppender possède deux classes filles : DailyRollingAppender, RollingFileAppender

Pour créer un appender, il suffit d'instancier un objet d'une de ces classes.

Tout comme les priorités, les appenders d'une categorie mère sont héritées par les categories filles. Pour éviter cette héritage par défaut, il faut mettre le champ additivity à false en utilisant la méthode setAdditivity() de la classe Category.

24.1.3. Les layouts

Ces composants représenté par la classe org.apache.log4j.Layout permettent de définir le format de la log. Un layout est associé à un appender lors de son instanciation.

Il existe plusieurs layouts définis par log4j :

- DateLayout
- HTMLLayout
- PatternLayout
- SimpleLayout
- XMLLayout

Le PatternLayout permet de préciser le format de la log grâce à des motifs qui sont dynamiquement remplacés par leur valeur à l'éxécution. Les motifs commencent par un caractère % suivi d'une lettre :

Motif	Role
%c	le nom de la catégorie qui a émis le message
%C	le nom de la classe qui a émis le message
%d	le timestamp de l'émission du message
%m	le message
%n	un retour chariot

%p	la priorité du message
%r	le nombre de milliseconde écoulé entre le lancement de l'application et l'émission du message
%t	le nom du thread
%x	NDC du thread
%%	le caractère %

Il est possible de préciser le formattage de chaque motif grâce à un alignement et/ou une troncature.Dans le tableau ci dessous, la caractère # représente une des lettres du tableau ci dessus, n représente un nombre de caractères.

Motif	Role
%#	aucun formattage (par défaut)
%n#	alignement à droite, des blancs sont ajoutés si la taille du motif est inférieure à n caractères
%−n#	alignement à gauche, des blanc sont ajoutés si la taille du motif est inférieure à n caractères
%.n	tronque le motif si il est supérieur à n caractères
%−n.n#	alignement à gauche, taille du motif obligatoirement de n caractères (troncature ou complément avec des blancs)

24.1.4. La configuration

Pour faciliter la configuration de log4j, l'API fournie plusieurs classes qui implémentent l'interface Configurator. La classe BasicConfgurator est le classe mère des classes PropertyConfigurator (pour la configuration via un fichier de propriétés) et DOMConfigurator (pour la configuration via un fichier XML).

La classe BasicConfigurator permet de configurer la catégorie root avec des valeurs par défaut. L'appel à la méthode configure() ajoute à la catégorie root la priorité DEBUG et un ConsoleAppender vers la sortie standard (System.out) associé à un PatternLayout (TTCC_CONVERSION_PATTERN qui est une constante défnie dans la classe PatternLayout).

Exemple :

```
import org.apache.log4j.*;

public class TestBasicConfigurator {
  static Category cat = Category.getInstance(TestBasicConfigurator.class.getName()) ;

  public static void main(String[] args) {
    BasicConfigurator.configure();
    cat.info("Mon message");
  }
}
```

Résultat :

```
0 [main] INFO TestBasicConfigurator  − Mon message
```

La classe PropertyConfigurator permet de configurer log4j à partir d'un fichier de propriétés ce qui évite la recompilation de classes pour modifier la configuration. La méthode configure() qui attend un nom de fichier permet de charger la

configuration.

Exemple :

```
import org.apache.log4j.*;

public class TestLogging6 {
   static Category cat = Category.getInstance(TestLogging6.class.getName());

   public static void main(String[] args) {
      PropertyConfigurator.configure("logging6.properties");
      cat.info("Mon message");
   }
}
```

Exemple : le fichier loggin6.properties de configuration de log4j

```
# Affecte a la categorie root la priorité DEBUG et un appender nommé CONSOLE_APP
log4j.rootCategory=DEBUG, CONSOLE_APP
# le appender CONSOL_APP est associé à la console
log4j.appender.CONSOLE_APP=org.apache.log4j.ConsoleAppender
# CONSOLE_APP utilise un PatternLayout qui affiche : le nom du thread, la priorité,
# le nom de la categorie et le message
log4j.appender.CONSOLE_APP.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE_APP.layout.ConversionPattern= [%t] %p %c − %m%n
```

Résultat :

```
C:\>java TestLogging6
[main] INFO TestLogging6 − Mon message
```

La classe DOMConfigurator permet de configurer log4j à partir d'un fichier XML ce qui évite aussi la recompilation de classes pour modifier la configuration. L méthode configure() qui attend un nom de fichier permet de charger la configuration. Cette méthode nécessite un parser XML de type DOM compatible avec l'API JAXP.

Exemple :

```
import org.apache.log4j.*;
import org.apache.log4j.xml.*;

public class TestLogging7 {
   static Category cat = Category.getInstance(TestLogging7.class.getName());

   public static void main(String[] args) {
      try {
         DOMConfigurator.configure("logging7.xml");
      } catch (Exception e) {
         e.printStackTrace();
      }
      cat.info("Mon message");
   }
}
```

Exemple : le fichier loggin7.xml de configuration de log4j

```
<?xml version="1.0" encoding="UTF−8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="CONSOLE_APP" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="[%t] %p %c − %m%n"/>
    </layout>
  </appender>
  <root>
```

```
    <priority value ="DEBUG" />
    <appender-ref ref="CONSOLE_APP" />
  </root>
</log4j:configuration>
```

Résultat :

```
C:\j2sdk1.4.0-rc\bin>java TestLogging7
[main] INFO TestLogging7 - Mon message
```

24.2. L'API logging

L'usage de fonctionnalités de logging dans les applications est tellement répendu que SUN a décidé de développer sa propre API et de l'intégrer au JDK à partir de la version 1.4.

Cette API a été proposée à la communauté sous la Java Specification Request numéro 047 (JSR-047).

Le but est de proposé un système qui puisse être exploitée facilement par toutes les applications.

L'API repose sur cinq classes principales et une interface:

- Logger : cette classe permet d'envoyer des messages sans le système de log
- LogRecord : cette classe encapsule le message
- Handler : cette classe représente la destination qui va recevoir les messages
- Formatter : cette classe permet de formatter le message avant son envoie vers la destination
- Filter : cette interface doit être implémentée par les classes dont le but est de déterminer si le message doit être envoyé vers une destination
- Level : cette représente le niveau de gravité du message

Un logger possède un ou plusieurs Handlers qui sont des entités qui vont recevoir les messages. Chaque handler peut avoir un filtre associé en plus du filtre associé au Logger.

Chaque message possède un niveau de sévérité représenté par la classe Level.

24.2.1. La classe LogManager

Cette classe est un singleton qui propose la méthode getLogManager() pour obtenir l'unique référence sur un objet de ce type.

Cette objet permet :

- de maintenir une liste de Logger désigné par un nom unique.
- de maintenir une liste de Handlers globaux
- de modifier le niveau de sévérité pris en compte pour un ou plusieurs Logger dont le début du nom est précisé

Pour réaliser ces actions, la classe LogManager possède plusieurs méthodes dont les principales sont :

Méthode	Rôle
void addGlobalHandler(Handler)	Ajoute un handler à la liste des handler globaux
Logger getLogger(String)	Permet d'ajouter un nouveau Logger dont le nom fourni en paramère lui sera attribué si il n'existe pas encore sinon la référence sur le Logger qui possède déjà ce nom est renvoyé.
void removeAllGlobalHandlers()	Supprime tous les Handler de la listede handler globaux

void removeGlobalHandler(Handler)	Supprime le Handler de la liste de Handler globaux
void setLevel(String, Level)	Permet de préciser le niveau de tout les Logger dont le nom commence par la chaine fournie en paramètre
LogManager getLogManager()	Renvoie l'instance unique de cette classe

Par défaut, la liste des Loggers contient toujours un Logger nommé global qui peut être facilement utilisé.

24.2.2. La classe Logger

La classe Logger est la classe qui se charge d'envoyer les messages dans la log. Un Logger est identifié par un nom qui est habituellement le nom qualifié de la classe dans laquelle le Logger est utilisé. Ce nom permet de gérer une hiérarchie de Logger. Cette gestion est assurée par le LogManager. Cette hiérarchie permet d'appliquer des modifications sur un Logger ainsi qu'à toute sa "descendance".

Il est aussi de possible de créer des Logger anonymes.

La méthode getLogger() de la classe LogManager permet d'instancier un nouvel objet Logger si aucun Logger possédant le nom passé en paramètre a déjà été défini sinon il renvoie l'instance existante.

La classe Logger se charge d'envoyer les messages aux Handlers enregistrés sous la forme d'un objet de type LogRecord. Par défaut, ces Handlers sont ceux enregistrés dans le LogManager. L'envoie des messages est conditionnés par la comparaison du niveau de sévérité de message avec celui associé au Logger.

La classe Logger possède de nombreuses méthodes pour générer des messages : plusieurs méthodes sont définis pour chaque niveau de sécurité. Plutôt que d'utiliser la méthode log() en précisant le niveau de sévérité, il est possible d'utiliser la méthode correspondante au niveau de sécurité.

Ces méthodes sont surchargées pour accepter plusieurs paramètres :

- le nom de la classe, le nom de la méthode et le message : les deux premières données sont très utiles pour faciliter le debuggage
- le message: dans ce cas, le framework tente de déterminer dynamiquement le nom de la classe et de la méthode

24.2.3. La classe Level

Chaque message est associé à un niveau de sévérité représenté par un objet de type Level. Cette classe définie7 niveaux de sévérité :

- Level.SEVERE
- Level.WARNING
- Level.INFO
- Level.CONFIG
- Level.FINE
- Level.FINER
- Level.FINEST

24.2.4. La classe LogRecord

24.2.5. La classe Handler

Le framework propose plusieurs classe filles qui représentent différents moyens pour émettre les messages :

- StreamHandler : envoie des messages dans un flux de sortie
- ConsoleHandler : envoie des messages sur la sortie standard d'erreur
- FileHandler : envoie des messages sur un fichier
- SocketHandler : envoie des messages dans une socket réseau
- MemoryHandler : envoie des messages dans un tampon en mémoire

24.2.6. La classe Filter

24.2.7. La classe Formatter

Le framework propose deux implémentations :

- SimpleFormatter : pour formatter l'enregistrement sous forme de chaine de caractères
- XMLFormatter : pour formatter l'enregistrement au format XML

XMLFormatter utilise un DTD particulière. Le tag racine est <log>. Chaque enregistrement est inclus dans un tag <record>.

24.2.8. Le fichier de configuration

Un fichier particulier au format Properties permet de préciser des paramètres de configuration pour le système de log tel que le niveau de sévérité géré par un Logger particulier et sa descendance, les paramètres de configuration des Handlers ...

Il est possible de préciser le niveau de sévérité pris en compte par tous les Logger :

.level = niveau

Il est possible de définir les handlers par défaut :

handlers = java.util.logging.FileHandler

Pour préciser d'autre handler, il faut les séparer par des virgules.

Pour préciser le niveau de sévérité d'un Handler, il suffit de le lui préciser :

java.util.logging.FileHandler.level = niveau

Un fichier par défaut est défini avec les autres fichiers de configuration dans le répertoire lib du JRE. Ce fichier ce nomme logging.properties.

Il est possible de préciser un fichier particulier précisant son nom dans la propriété système java.util.logging.config.file

exemple : java −Djava.util.logging.config.file=monLogging.properties

24.2.9. Exemples d'utilisation

Cette section est en cours d'écriture

24.3. D'autres API de logging

Il existe d'autres API de logging dont voici une liste non exhaustive :

Produit	URL
Lumberjack	http://javalogging.sourceforge.net/
Javalog	http://sourceforge.net/projects/javalog/
Jlogger de Javelin Software	http://www.javelinsoft.com/jlogger/

Partie 4 : développement serveur

Cette quatrième partie traite d'une utilisation de java en forte expansion : le développement côté serveur. Ce type de développement est poussée par l'utilisation d'internet notamment.

Ces développements sont tellements important que Sun propose une véritable plate−forme, basée sur le J2SE et orienté entreprise dont les API assurent le développement côté serveur : J2EE (Java 2 Entreprise Edition).

Cette partie regroupe plusieurs chapitres :

- J2EE : introduit la plate−forme Java 2 Entreprise Edition
- les servlets : plonge au coeur de l'API servlet qui est un des composants de base pour le développement d'applications Web
- les JSP : poursuit la discusion avec les servlets en explorant un mécanisme basé sur celles ci pour réaliser facilement des pages web dynamiques
- XML : présente XML qui est un technologie tendant à s'imposer pour les échanges de données et explore les API java pour utiliser XML
- JNDI : introduit l'API capable d'acceder aux services de nommage et d'annuaires
- JMS : indique comment utiliser cette API qui permet l'utilisation de système de messages pour l'échange de données entre applications
- Java Mail : traite de l'API qui permet l'envoie et la réception d'e mail
- EJB : propose une présentation del'API et les spécifications pour des objets chargés de contenir les règles métiers
- les services web :

25. J2EE

J2EE est l'acronyme de Java 2 Entreprise Edition. Cette édition est dédiée à la réalisation d'applications pour entreprises. J2EE est basée sur J2SE (Java 2 Standard Edition) qui contient les API de bases de java.

J2EE est une plate−forme fortement orientée serveur pour le développement et l'éxécution d'applications distribuées. Elle est composée de deux parties essentielles :

- un ensemble de spécifications pour une infrastructure dans laquelle s'éxécute les composants écrits en java : un tel environnement se nomme serveur d'application.
- un ensemble d'API qui peuvent être obtenues et utilisées séparement. Pour être utilisées, certaines nécessitent une implémentation de la part d'un fournisseur tiers.

Sun propose une implémentation minimale des spécifications de J2EE. Cette implémentation permet de développer des applications respectant les spécifications mais n'est pas prévue pour être utilisée dans un environnement de production. Ces spécifications doivent être respectées par les outils développés par des éditeurs tiers.

Ce chapitre est en cours d'écriture

25.1. Les API de J2EE

J2EE regroupe un ensemble d'API pour le développement d'applications d'entreprise.

API	Rôle	version de l'API dans J2EE 1.2	version de l'API dans J2EE 1.3
Entreprise java bean (EJB)	Composants serveurs contenant la logique métier	1.1	2.0
Remote Méthod Invocation (RMI) et RMI−IIOP	RMI permet d'utilisation d'objet java distribué. RMI−IIOP est une extension de RMI pour utiliser avec CORBA.	1.0	
Java Naming and Directory Interface (JNDI)	Accès aux services de nommage et aux annuaires d'entreprises	1.2	1.2
		2.0	2.0

Java Database Connectivity (JDBC)	Accès aux bases de données. J2EE intègre une extension de cette API		
Java Transaction API (JTA) Java Transaction Service (JTS)	Support des transactions	1.0	1.0
Java Messaging service (JMS)	Support de messages via des MOM (Messages Oriented Middleware)	1.0	1.0
Servlets	Composants basé sur le concept C/S pour ajouter des fonctionnalités à un serveur. Pour le moment, principalement utilisé pour étendre un serveur web	2.2	2.3
JSP		1.1	1.2
Java IDL	Utilisation de CORBA		
JavaMail	Envoie et réception d'email	1.1	1.2
J2EE Connector Architecture (JCA)	Connecteurs pour accéder à des ressources de l'entreprises tel que CICS, TUXEDO, SAP ...		1.0
Java API for XML Parsing (JAXP)	Analyse et exploitation de données au format XML		1.1
Java Authentication and Authorization Service (JAAS)	Echange sécurisé de données		1.0
JavaBeans Activation Framework	Utilisé par JavaMail : permet de déterminer le type mime		1.2

Ces API peuvent être regroupées en trois grandes catégories :

- les composants : Servlet, JSP, EJB
- les services : JDBC, JTA/JTS, JNDI, JCA, JAAS
- la communication : RMI−IIOP, JMS, Java Mail

25.2. L'environnement d'éxécution des applications J2EE

J2EE propose des spécifications pour une infrastructure dans laquelle s'éxécute les composants. Ces spécifications décrivent les rôles de chaque éléments et précisent un ensemble d'interfaces pour permettre à chacun de ces éléments de communiquer.

Ceci permet de séparer les applications et l'environnement dans lequel il s'éxécute. Les spécifications précisent à l'aide des API un certain nombre de fonctionnalités que doivent implémenter l'environnement d'exécution. Ces fonctionnalités sont de bas niveau et permettent aux développeurs de se concentrer sur la logique métier.

Pour éxecuter ces composants de natures différentes, J2EE défini des conteneurs pour chacun de ces composants. Il définit pour chaque composants des interfaces qui leur permettront de dialoguer avec les composants lors de leur execution. Les conteneurs permettent aux applications d'accéder aux ressources et aux services en utilisant les API.

Les appels aux applications se font par des clients via les conteneurs. Les clients n'accèdent pas directement aux applications mais sollicite le conteneur.

25.2.1. Les conteneurs

Les conteneurs assurent la gestion du cycle de vie des composants qui s'éxécutent en eux.

Les conteneurs fournissent des services qui peuvent être utilisés par les applications lors de leur éxécution.

Pour déployer une application dans un conteneur, il faut lui fournir deux éléments :

- l'application avec tous les composants (classes compilées, ressources ...) regroupée dans une archive ou module. Chaque conteneur possède son propre format d'archive.
- un fichier descripteur de déploiement contenu dans le module qui précise au conteneur des options pour éxécuter l'application

Il existe trois types d'archive :

Archive / module	Contenu	Extension	Descripteur de déploiement
java	Regroupe des classes	jar	application−client.jar
web	Regroupe les servlets et les JSP ainsi que les ressources necessaires à leur execution (classes, bibliothèque de balises, images, ...)	war	web.xml
EJB	Regroupe les EJB et leur composants (classes)	jar	ejb−jar.xml

Une application est un regroupement d'une ou plusieurs modules dans un fichier EAR (Entreprise ARchive).
L'application est décrite dans un fichier application.xml lui même contenu dans le fichier EAR

Il existe autant de conteneurs que de type d'applications :

- conteneur web : pour éxécuter les servlets et les JSP
- conteneur d'EJB : pour éxécuter les EJB

25.2.2. Le conteneur web

L'implémentation de référence pour ce type de conteneur est le projet open source Tomcat du groupe Apache.

25.2.3. Le conteneur d'EJB

25.3. L'assemblage et le déploiement d'applications J2EE

J2EE propose une spécification pour décrire le mode d'assemblage et de déploiement d'une application J2EE.

Une application J2EE peut regrouper différents modules : modules web, modules EJB ... Chacun de ces modules possède son propre mode de packaging. J2EE propose de regrouper ces différents modules dans un module unique sous la forme d'un fichier EAR (Entreprise ARchive).

Le format de cet archive est très semblable à celui des autres archives :

- un contenu : les différents modules qui composent l'application (module web, EJB, fichier RAR ...)
- un fichier descripteur de déploiement

Les serveurs d'application extraient chaque modules du fichier EAR et les déploient séparément un par un.

25.3.1. Le contenu et l'organisation d'un fichier EAR

Le fichier EAR est composé au minimum :

- d'un ou plusieurs modules
- un répertoire META−INF contenant un fichier descripteur de déploiement nommé application.xml

Les modules ne doivent pas obligatoirement être insérés à la racine du fichier EAR : ils peuvent être mis dans un des sous répertoires pour organiser le contenu de l'application. Il est par exemple pratique de créer un répertoire lib qui contient les fichier .jar des bibliothèques communes aux différents modules.

25.3.2. La création d'un fichier EAR

Pour créer un fichier EAR, il est possible d'utiliser un outils graphique fourni par le vendeur du serveur d'application ou créer le fichier manuellement en suivant les étapes suivantes :

1. créer l'arboresence des répertoires qui vont contenir les modules
2. insérer dans cette arboresence les différents modules à inclure dans le fichier EAR
3. créer le répertoire META−INF (en respectant la casse)
4. créer le fichier application.xml dans ce répertoire
5. utiliser l'outils jar pour créer le fichier le fichier EAR en précisant les options cvf, le nom du fichier ear avec son extension et les différents éléments qui compose le fichier (modules, répertoire dont le répertoire META−INF).

25.3.3. Les limitations des fichiers EAR

Actullement les fichiers EAR ne servent à regrouper différents modules pour former une seule entité. Rien n'est actuellement prévu pour prendre en compte la configuration des objets permettant l'accès aux ressources par l'application tel qu'une base de données (JDBC pour DataSource, pool de connexion ...), un système de message (JMS), etc ...

Pour lever une partie de ces limites, les serveurs d'applications commerciaux proposent souvent des mécanismes propriétaires supplémentaires pour palier à ces manques en attendant une évolution des spécifications.

Ce chapitre est en cours d'écriture

26. Les servlets

Les serveurs web sont de base uniquement capable de renvoyer des fichiers présents sur le serveur en réponse à une requête d'un client. Cependant, pour permettre l'envoie d'une page HTML contenant par exemple une liste d'articles répondant à différents critères, il faut créer dynamiquement ces pages HTML. Plusieurs solutions existent pour ces traitements. Les servlets java sont une des ces solutions.

Mais les servlets peuvent aussi servir à d'autres usages.

Sun fourni des informations sur les servlets sur son site : http://java.sun.com/products/servlet/index.html

26.1. Présentation des servlets

Une servlet est un programme qui s'éxécute côté serveur en tant qu'extension du serveur. Elle reçoit une requête du client, elle effectue des traitements et renvoie le résultat. La liaison entre la servlet et le client peut être directe ou passer par un intermédiaire comme par exemple un serveur http.

Même si pour le moment la principale utilisation des servlets est la génération de pages html dynamiques utilisant le protocole http et donc un serveur web, n'importe quel protocole reposant sur le principe de requête/réponse peut faire usage d'une servlet.

Ecrite en java, une servlet en retire ses avantages : la portabilité, l'accès à toutes les API de java dont JDBC pour l'accès aux bases de données, ...

Une servlet peut être invoquée plusieurs fois en même temps pour répondre à plusieurs requêtes simultanées.

La servlet se positionne dans une architecture Client/Serveur trois tiers dans le tiers du milieu entre le client léger chargé de l'affichage et la source de données.

26.1.1. Le fonctionnement d'une servlet (cas d'utilisation de http)

Un serveur d'application permet de charger et d'éxécuter les servlets dans une JVM. C'est une extension du serveur web. Ce serveur d'application contient entre autre un moteur de servlets qui se charge de manager les servlets qu'il contient.

Pour éxécuter une servlet, il suffit de saisir une URL qui désigne la servlet dans un navigateur.

1. Le serveur reçoit la requête http qui necessite une servlet de la part du navigateur
2. Si c'est la premiere sollicitation de la servlet, le serveur l'instancie. Les servlets sont stockées (sous forme de fichier .class) dans un répertoire particulier du serveur. Ce répertoire dépend du serveur d'application utilisé. La servlet reste en mémoire jusqu'à l'arret du serveur. Certains serveurs d'application permettent aussi d'instancier des servlets dès le lancement du serveur.

   La servlet en mémoire, peut être appelée par plusieurs threads lancés par le serveur pour chaque requête. Ce principe de fonctionnement évite d'instancier un objet de type servlet à chaque requête et permet de maintenir un ensemble de ressources actives tel qu'une connection à une base de données.

3. le serveur crée un objet qui représente la requète http et objet qui contiendra la réponse et les envoie à la servlet
4. la servlet créé dynamiquement la reponse sous forme de page html transmise via un flux dans l'objet contenant la réponse. La création de cette réponse utilise bien sûre la requète du client mais aussi un ensemble de ressources incluses sur le serveur tels de que des fichiers ou des bases de données.
5. le serveur récupère l'objet réponse et envoie la page html au client.

26.1.2. Les outils nécessaires pour développer des servlets

Pour développer des servlets avec le JDK standard édition, il faut utiliser le Java Server Development Kit (JSDK) qui est une extension du JDK.

Pour réaliser les tests, le JSDK fournit, dans sa version 2.0 un outils nommé servletrunner et depuis sa version 2.1, il fournit un serveur http allégé.

Pour une utilisation plus poussée ou une mise en exploitation des servlets, il faut utiliser un serveur d'application : il existe de nombreuses versions commerciales tel que IBM Webpshere ou BEA WebLogic mais aussi des versions libres tel que Tomcat du projet GNU Jacarta.

Ce serveur d'application doit utiliser ou inclure un serveur http dont le plus utilisé est apache.

Le choix d'un serveur d'application doit tenir compte de la version du JSDK qu'il supporte pour être compatible avec celle utilisée pour le développement des servlets. Le choix entre un serveur commercial et un libre doit tenir compte principalement du support technique, des produits annexes fournis et des outils d'installation et de configuration.

Pour simplement développer des servlets, le choix d'un serveur libre se justifie pleinenement de part leur gratuité et leur « légèreté ».

26.1.3. Le role du serveur d'application

Dans le serveur d'application tourne un moteur de servlet qui prend en charge et gère les servlets : chargement de la servlet, gestion de son cycle de vie, passage des requètes et des réponses ...

Le chargement et l'instanciation d'une servlet se font selon le paramétrage soit au lancement du serveur soit à la première invocation de la servlet. Dès l'instanciation, la servlet est initialisée une seul et unique fois avant de pouvoir répondre aux requètes. Cette initialisation peut permettre de mettre en place l'accès à des ressources tel qu'une base de données.

26.1.4. Les différences entre les servlets et les CGI

Les programmes ou script CGI (Common Gateway Interface) sont aussi utilisés pour générer des pages HTML dynamiques. Ils représentent la plus ancienne solution pour réaliser cette tache.

Un CGI peut être écrit dans de nombreux langages.

Il existe plusieurs avantages à utiliser des servlets plutôt que des CGI :

- la portabilité offerte par Java bien que certain langage de script tel que PERL tourne sur plusieurs plateformes.
- la servlet reste en mémoire une fois instanciée ce qui permet de garder des ressources systèmes et gagner le temps de l'initialisation. Un CGI est chargé en mémoire à chaque requète, ce qui réduit les performances.
- les servlets possède les avantages de toutes les classes écrites en java : accès aux API, aux java beans, le garbage collector ...

26.2. L'API servlet

Les servlets sont conçues pour agir selon un modele de requète/reponse. Tous les protocoles utilisant ce modèle peuvent être utilisé tel que http, ftp, etc ...

L'API servlets est une extension du jdk de base, et en tant que tel elle est regroupée dans des packages préfixés par javax

L'API servlet regroupe un ensemble de classes dans deux packages :

- javax.servlet : contient les classes pour développer des serlvets génériques indépendantes d'un protocole
- javax.servlet.http : contient les clases pour développer des servlets qui repose sur le protocole http utilisé par les serveurs web.

Le package javax.servlet définit plusieurs interfaces, méthodes et exceptions :

javax.servlet	Nom	Role
Les interfaces	RequestDispatcher	Définition d'un objet qui permet le renvoie d'une requète vers une autre ressource du serveur (une autre servlet, une JSP ...)
	Servlet	Définition de base d'une servlet
	ServletConfig	Définition d'un objet pour configurer la servlet
	ServletContext	Définition d'un objet pour obtenir des informations sur le contexte d'execution de la servlet
	ServletRequest	Définition d'un objet contenant la requète du client
	ServletResponse	Définition d'un objet qui contient la reponse renvoyée par la servlet
	SingleThreadModel	Permet de définir une servlet qui ne répondra qu'à une seule requète à la fois
Les classes	GenericServlet	Classe définissant une servlet indépendante de tout protocole
	ServletInputStream	Flux permet la lecteur des donnes de la requète cliente
	ServletOutPutStream	Flux permettant l'envoie de la reponse de la servlet
Les exceptions	SevletException	Exception generale en cas de problème de la servlet
	UnavailableException	Exception levée si la servlet n'est pas disponible

Le package javax.servlet.http définit plusieurs interfaces et méthodes :

Javax.servlet	Nom	Role
Les interfaces	HttpServletRequest	Hérite de ServletRequest : définie un objet contenant une requète selon le protocole http
	HttpServletResponse	Hérite de ServletResponse : définie un objet contenant la reponse de la servlet selon le protocole http
	HttpSession	Définie un objet qui représente une session
Les classes	Cookie	Classe représentant un cookie (ensemble de données sauvegardées par le brower sur le poste client)
	HttpServlet	

		Hérite de GenericServlet : classe définissant une servlet utilisant le protocole http
	HttpUtils	Classe proposant des méthodes statiques utiles pour le développement de servlet http

26.2.1. L'interface Servlet

Une servlet est une classe Java qui implémente l'interface javax.servlet.Servlet. Cette interface définie 5 méthodes qui permettent au serveur de dialoguer avec la servlet : elle encapsule ainsi les méthodes nécessaire à la communication entre le serveur et la servlet.

Méthode	Rôle
void service (ServletRequest req, ServletResponse res)	Cette méthode est éxécutée par le serveur lorsque la servlet est sollicitée : chaque requète du client déclenche une seule execution de cette méthode. Cette méthode pouvant être éxécutée par plusieurs threads, il faut prévoir un processus d'exclusion pour l'utilisation de certaines ressources.
void init(ServletConfig conf)	Initialisation de la servlet. Cette méthode est appelée une seule fois après l'instanciation de la servlet. Aucun traitement ne peut être effectué par la servlet tant que l'éxecution de cette méthode n'est pas terminée.
ServletConfig getServletConfig()	Renvoie l'objet ServletConfig passé à la méthode init
void destroy()	Cette méthode est appelé lors la destruction de la servlet. Elle permet de liberer proprement certaines ressources (fichiers, bases de données ...). C'est le serveur qui appelle cette méthode.
String getServletInfo()	Renvoie des informations sur la servlet.

Les méthodes init(), service() et destroy() assure le cycle de vie de la servlet en étant respectivement appellée lors de la création de la servlet, lors de son appel pour le traitement d'une requète et lors de sa destruction.

La méthode init() est appellée par le serveur juste après l'instanciation de la servlet.

La méthode service() ne peut pas être invoquée tant que la méthode init n'est pas terminée.

La méthode destroy() est appelée juste avant que le serveur ne détruise la servlet : cela permet de liberer des ressources allouées dans la méthode init() tel qu'un fichier ou une connection à une base de données.

26.2.2. La requète et la réponse

L'interface ServletRequest définit plusieurs méthodes qui permettent d'obtenir les données sur la requète du client:

Méthode	Role
ServletInputStream getInputStream()	Permet d'obtenir un flux pour les données de la requète
BufferedReader getReader()	Idem

L'interface ServletResponse définit plusieurs méthodes qui permettent de fournir la réponse faite par la servlet suite à ces traitements :

Méthode	Role
SetContentType	Permet de préciser le type MIME de la réponse
ServletOutputStream getOutputStream()	Permet d'obtenir un flux pour envoyer la réponse
PrintWriter getWriter()	Permet d'obtenir un flux pour envoyer la réponse

26.2.3. Un exemple de servlet

Une servlet qui implémente simplement l'interface Servlet doit évidemment redéfinir toute les méthodes définies dans l'interface.

Il est très utile lorsque que l'on créé une servlet qui implémente directement l'interface Servlet de sauvegarder l'objet ServletConfig fourni par le serveur en paramètre de la méthode init() car c'est le seul moment ou l'on a accès à cet objet.

Exemple (code java 1.1) :

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

public class TestServlet implements Servlet {
  private ServletConfig cfg;

  public void init(ServletConfig config) throws ServletException {
     cfg = config;
  }

  public ServletConfig getServletConfig() {
    return cfg;
  }

  public String getServletInfo() {
    return "Une servlet de test";
  }

  public void destroy() {
  }

  public void service (ServletRequest req,  ServletResponse res )
  throws ServletException, IOException  {
    res.setContentType( "text/html" );
    PrintWriter out = res.getWriter();
    out.println( "<THML>" );
    out.println( "<HEAD>");
    out.println( "<TITLE>Page generee par une servlet</TITLE>" );
    out.println( "</HEAD>" );
    out.println( "<BODY>" );
    out.println( "<H1>Bonjour</H1>" );
    out.println( "</BODY>" );
    out.println( "</HTML>" );
    out.close();
  }
}
```

26.3. Le protocole HTTP

Le protocol HTTP est un protocol qui fonctionne sur le modèle client/serveur. Un client qui est un application (souvent un navigateur web) envoie une requete à un serveur (un serveur web). Ce serveur attend en permanence les requètes sur un port particulier (par défaut le port 80). A la reception de la requete, le serveur lance un thread qui va la traiter pour générer la réponse. Le serveur envoie la réponse au client.

Une particularité du protocol HTTP est de maintenir la connexion entre le client et le serveur uniquement durant l'échange de la requète et de la réponse.

Il existe deux versions principales du protocole HTTP : 1.0 et 1.1.

La requête est composé de trois parties :

- la commande
- la section en−tête
- le corps

la première ligne de la requète contient la commande à executer par le serveur. La commande est suivi éventuellement d'un argument qui précise la commande (par exemple l'url de la ressource demandée). Enfin la ligne doit contenir la version du protocole HTTP utilisée précédé de HTTP/.

Exemple :

GET / index.html HTTP/1.0

Avec HTTP 1.1, les commandes suivantes sont définies : GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE et CONNECT. Les trois premières sont les plus utilisées.

Il est possible de fournir sur les lignes suivantes de la partie en−tête des paramètres supplémentaires. Cet partie en−tête est optionnelles. Les informations fournies peuvent permettre au serveur d'obtenir des informations sur le client. Chaque information doit être mise sur une ligne unique. Le format est nom_du_champ:valeur. Les champs sont prédéfinis et sont sensibles à la casse.

Une ligne vide doit précéder le corps de la requête. Le contenu du corps de la requête dépend du type de la commande.

La requète doit obligatoirement être terminé par une ligne vide.

La réponse est elle aussi composée des trois mêmes parties :

- une ligne de status
- une en−tête dont le contenu est normalisé
- un corps dont le contenu dépend totalement de la requête

La première ligne de l'en−tête contient un état qui est composé : de la version du protocole HTTP utilisé, du code de status et d'une description succinte de ce code.

Le code de status composé de trois chiffres renseigne sur le résultat du traitement qui a généré cette réponse. Ce code peut être regroupé en plusieurs catégories en fonction de leur valeur :

Plage de valeur du code	Signification
100 à 199	Information
200 à 299	traitement avec succès
300 à 399	la requete a été redirigé
400 à 499	la requete est incomplete ou erronée
500 à 599	Une erreur est intervenue sur le serveur

Plusieurs codes son définis par le protocole HTTP dont les plus importants sont :

- 200 : traitement correct de la requète

- 204 : traitement correct de la requète mais la réponse ne contient aucun contenu (ceci permet au browser de laisser la page courante affichée)
- 404 : la ressource demande n'est pas trouvée (surement le plus célèbre)
- 500 : erreur interne du serveur

L'en−tête contient des informations qui précise le contenu de la réponse.

Le corps de la réponse est précédé par une ligne vide.

Cette section est en cours d'écriture

26.4. Les servlets http

L'usage principale des servlets est la création de pages HTML dynamiques. Sun fourni une classe qui encapsule une servlet utilisant le protocole http. Cette classe est la classe HttpServlet

Cette classe hérite de GenericServlet, donc elle implémente l'interface Servlet, et redéfinie toutes les méthodes nécessaires pour fournir un niveau d'abstraction permettant de développer facilement des servlets avec le protocole http.

Ce type de servlet n'est pas utile que pour générer des pages HTML bien que cela soit son principale usage, elle peut aussi réaliser un ensemble de traitements tel que mettre à jour une base de données. En réponse, elle peut générer une page html qui indique le succès ou non de la mise à jour.

Elle définit un ensemble de fonctionnalités très utiles : par exemple, elle contient une méthode service() qui appelle certaines méthodes à redéfinir en fonction du type de requète http (doGet(), doPost(), etc ...).

La requète du client est encapsulée dans un objet qui implémente l'interface HttpServletRequest : cette objet contient les données de la requète et des informations sur le client.

La réponse de la servlet est encapsulée dans un objet qui implémente l'interface HttpServletResponse.

Typiquement pour définir une servlet, il faut définir une classe qui hérite de la classe HttpServlet et redéfinir la classe doGet et/ou doPost selon les besoins.

La méthode service héritée de HttpServlet appelle l'une ou l'autre de ces méthodes en fonction du type de la requète http :

- une requète GET : c'est une requète qui permet au client de demander une page html
- une requète POST : c'est une requète qui permet au client d'envoyer des informations issues par exemple d'un formulaire

Une servlet peut traiter un ou plusieurs types de requètes grace à plusieurs autres méthodes :

- doHead : pour les requètes http de type HEAD
- doPut : pour les requètes http de type PUT
- doDelete : pour les requètes http de type DELETE
- doOptions : pour les requètes http de type OPTIONS
- doTrace : pour les requètes http de type TRACE

La classe HttpServlet hérite aussi de plusieurs méthodes définies dans l'interface Servlet : init, destroy et getServletInfo.

26.4.1. La méthode init()

Si cette méthode doit être redéfinie, il est important d'invoquer la méthode héritée avec un super.init(config), config étant l'objet fourni en paramètre de la méthode. La méthode définie dans la classe HttpServlet sauvegarde l'objet de type ServletConfig.

De plus, le classe GenericServlet implémente l'interface ServletConfig. Les méthodes redéfinies pour cette interface utilisent l'objet sauvegardé. Ainsi, la servlet peut utiliser sa propre méthode getInitParameter() ou utiliser la méthode getInitParameter() de l'objet de type ServletConfig. La première solution permet un usage plus facile dans toute la servlet.

Sans l'appel à la méthode héritée lors d'une redéfinition, la méthode getInitParameter() de la servlet levera une exception de type NullPointerException.

26.4.2. L'analyse de la requète

La méthode service() est la méthode qui est appelée lors d'un appel à la servlet.

Par défaut dans la classe HttpServlet, cette méthode contient du code qui réalise une analyse de la requète client contenue dans l'objet HttpServletRequest. Selon le type de requète GET ou POST, elle appelle la méthode doGet() ou doPost(). C'est bien ce type de requète qui indique quelle méthode utiliser dans la servlet.

Ainsi, la méthode service() n'est pas à redéfinir pour ces requètes et il suffit de redéfinir les méthodes doGet() et/ou doPost() selon les besoins.

26.4.3. La méthode doGet()

Une requète de type GET est utile avec des liens. Par exemple :

```
<A HREF="http://localhost:8080/examples/servlet/tomcat1.MyHelloServlet">test de la servlet</A>
```

Dans une servlet de type HttpServlet, une telle requète est associée à la méthode goGet.

La signature de la méthode doGet() :

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException
{

}
```

Le traitement typique de la méthode doGet() est d'analyser les paramètres de la requète, alimenter les données de l'en−tête de la réponse et d'écrire la réponse.

26.4.4. La méthode doPost()

Un requète POST n'est utilisable qu'avec un formulaire HTML.

Exemple de code HTML :

```
<FORM ACTION="http://localhost:8080/examples/servlet/tomcat1.TestPostServlet"
METHOD="POST">
<INPUT NAME="NOM">
<INPUT NAME="PRENOM">
<INPUT TYPE="ENVOYER">
```

```
</FORM>
```

Dans l'exemple ci dessus, le formulaire comporte deux zones de saisies correspondant à deux paramètres : NOM et PRENOM.

Dans une servlet de type HttpServlet, une telle requète est associée à la méthode doPost().

La signature de la méthode doPost() :

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws IOException
{

}
```

La méthode doPost() doit généralement recueillir les paramètres pour les traiter et générer la réponse. Pour obtenir la valeur associée à chaque paramètre il faut utiliser la méthode getParameter de l'objet HttpServletRequest. Cette méthode attent en paramètre le nom du paramètre dont on veut la valeur. Ce paramètre est sensible à la casse.

Exemple :

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException
{
   String nom = request.getParameter("NOM");
   String prenom = request.getParameter("PRENOM");
}
```

26.4.5. La génération de la réponse

La servlet envoie sa réponse au client en utilisant un objet de type HttpServetResponse. HttpServletResponse est une interface : il n'est pas possible d'instancier un tel objet mais le moteur de servlet instancie un objet qui implémente cette interface et le passe en paramètre de la méthode service.

Cette interface possède plusieurs méthodes pour mettre à jour l'en−tête http et le page HTML de retour.

Méthode	Rôle
void sendError (int)	Envoie une erreur avec un code retour et un message par défaut
void sendError (int, String)	Envoie une erreur avec un code retour et un message
void setContentType(String)	Héritée de ServletResponse, cette méthode permet de préciser le type MIME de la réponse
void setContentLength(int)	Héritée de ServletResponse, cette méthode permet de préciser la longueur de la réponse
ServletOutputStream getOutputStream()	Héritée de ServletResponse, elle retourne un flux pour l'envoie de la réponse
PrintWriter getWriter()	Héritée de ServletResponse, elle retourne un flux pour l'envoie de la réponse

Avant de générer la réponse sous forme de page HTML, il faut indiquer dans l'en tête du message http, le type mime du contenu du message. Ce type sera souvent « text/html » qui correspond à une page HTML mais il peut aussi prendre d'autre valeur en fonction de ce que retourne la servlet (une image par exemple). La méthode à utiliser est

setContentType.

Il est aussi possible de préciser la longueur de la réponse avec la méthode setContentLength(). Cette précision est optionnelle mais si elle est utilisée, la longueur doit être exacte pour éviter des problèmes.

Il est préférable de créer une ou plusieurs méthodes recevant en paramètre l'objet HttpServletResponse qui seront dédiées à la génération du code HTML afin de ne pas alourdir les méthodes doXXX().

Il existe plusieurs façons de générer une page HTML : elles utiliseront toutes soit la méthode getOutputStream() ou getWriter() pour obtenir un flux dans lequel la réponse sera envoyée.

- Utilisation d'un StringBuffer et getOutputStream

Exemple (code java 1.1) :

```
protected void GenererReponse1(HttpServletResponse reponse) throws IOException
{
   //creation de la reponse
   StringBuffer sb = new StringBuffer();
   sb.append("<HTML>\n");
   sb.append("<HEAD>\n");
   sb.append("<TITLE>Bonjour</TITLE>\n");
   sb.append("</HEAD>\n");
   sb.append("<BODY>\n");
   sb.append("<H1>Bonjour</H1>\n");
   sb.append("</BODY>\n");
   sb.append("</HTML>");

   // envoie des infos de l'en tete
   reponse.setContentType("text/html");
   reponse.setContentLength(sb.length());

   // envoie de la reponse
   reponse.getOutputStream().print(sb.toString());
}
```

L'avantage de cette méthode est qu'elle permet facilement de déterminer la longueur de la réponse.

Dans l'exemple, l'ajout des retours chariot '\n' à la fin de chaque ligne n'est pas obligatoire mais elle facilite la compréhension du code HTML surtout si il devient plus complexe.

- Utilisation directe de getOutputStream

Exemple :

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

public class TestServlet4 extends HttpServlet {

   public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException {
      res.setContentType("text/html");
      ServletOutputStream out = res.getOutputStream();
      out.println("<HTML>\n");
      out.println("<HEAD>\n");
      out.println("<TITLE>Bonjour</TITLE>\n");
      out.println("</HEAD>\n");
      out.println("<BODY>\n");
      out.println("<H1>Bonjour</H1>\n");
      out.println("</BODY>\n");
      out.println("</HTML>");
   }
```

```
}
```

- Utilisation de la méthode getWriter()

Exemple (code java 1.1) :

```
protected void GenererReponse2(HttpServletResponse reponse) throws IOException {

   reponse.setContentType("text/html");

   PrintWriter out = reponse.getWriter();

   out.println("<HTML>");
   out.println("<HEAD>");
   out.println("<TITLE>Bonjour</TITLE>");
   out.println("</HEAD>");
   out.println("<BODY>");
   out.println("<H1>Bonjour</H1>");
   out.println("</BODY>");
   out.println("</HTML>");
}
```

Avec cette méthode, il faut préciser le type MIME avant d'écrire la réponse. L'emploi de la méthode println() permet d'ajouter un retour chariot en fin de chaque ligne.

Si un problème survient lors de la génération de la réponse, la méthode sendError() permet de renvoyer une erreur au client : un code retour est positionné dans l'en tête http et le message est indiqué dans une simple page HTML.

26.4.6. Un exemple de servlet HTTP très simple

Toute servlet doit au moins importer trois packages : java.io pour la gestion des flux et deux packages de l'API servlet ; javax.servlet.* et javax.servlet.http.

Il faut déclarer une nouvelle classe qui hérite de HttpServlet.

Il faut redéfinir la méthode doGet() pour y insérer le code qui va envoyer dans un flux le code HTML de la page générée.

Exemple (code java 1.1) :

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

public class MyHelloServlet extends HttpServlet {

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

      response.setContentType("text/html");

      PrintWriter out = response.getWriter();

      out.println("<html>");
      out.println("<head>");
      out.println("<title>Bonjour tout le monde</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<h1>Bonjour tout le monde</h1>");
      out.println("</body>");
      out.println("</html>");
   }
```

```
}
```

La méthode getWriter() de l'objet HttpServletResponse renvoie un flux de type PrintWriter dans lequel on peut écrire la réponse.

Si aucun traitement particulier n'est associé à une requète de type POST, il est pratique de demander dans la méthode doPost() d'éxécuter la méthode doGet(). Dans ce cas, la servlet est capable de renvoyer une réponse pour les deux types de requète.

Exemple (code java 1.1) :

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

   this.doGet(request, response);

}
```

26.5. Les informations sur l'environnement d'execution des servlets

Une servlet est éxécutée dans un contexte particulier mis en place par le moteur de servlet.

La servlet peut obtenir des informations sur ce contexte.

La servlet peut aussi obtenir des informations à partir de la requète du client.

26.5.1. Les paramètres d'initialisation

Dès que de la servlet est instanciée, le moteur de servlet appelle sa méthode init() en lui donnant en paramètre un objet de type ServletConfig.

ServletConfig est une interface qui possède deux méthodes permettant de connaître les paramètres d'initialisation :

- String getInitParameter(String);

  Exemple :

  ```
  String param;

  public void init(ServletConfig config) {

     param = config.getInitParameter("param");

  }
  ```

- Enumeration getInitParameterNames() : retourne une enumeration des paramètres d'initialisation

  Exemple (code java 1.1) :

  ```
  public void init(ServletConfig config) throws ServletException {

     cfg = config;

     System.out.println("Liste des parametres d'initialisation");

     for (Enumeration e=config.getInitParameterNames(); e.hasMoreElements();) {
  ```

```
      System.out.println(e.nextElement());
   }
}
```

La déclaration des paramètres d'initialisation dépend du serveur qui est utilisé.

26.5.2. L'objet ServletContext

La servlet peut obtenir des informations à partir d'un objet ServletContext retourné par la méthode getServletContext() d'un objet ServletConfig.

Il est important de s'assurer que cet objet ServletConfig, obtenu par la méthode init() est soit explicitement sauvegardé soit sauvegardé par l'appel à la méthode init() héritée qui effectue cette sauvegarde.

L'interface ServletContext contient plusieurs méthodes dont les principales sont :

méthode	Role	Deprecated
String getMimeType(String)	Retourne le type MIME du fichier en paramètre	
String getServletInfo()	Retourne le nom et le numero de version du moteur de servlet	
Servlet getServlet(String)	Retourne une servlet à partir de son nom grace au context	Ne plus utiliser depuis la version 2.1 du jsdk
Enumeration getServletNames()	Retourne une enumeration qui contient la liste des servlets realatives au contexte	Ne plus utiliser depuis la version 2.1 du jsdk
void log(Exception, String)	Ecrit les informations fournies en paramètre dans le fichier log du serveur	Utiliser la nouvelle méthode surchargée de log()
void log(String)	Idem	
void log (String, Throwable)	Idem	

Exemple : écriture dans le fichier log du serveur :

```
public void init(ServletConfig config) throws ServletException {
   ServletContext sc = config.getServletContext();
   sc.log( "Demarrage servlet TestServlet" );
}
```

Le format du fichier log est dependant du serveur utilisé :

Exemple : résultat avec tomcat

```
Context log path="/examples" :Demarrage servlet TestServlet
```

26.5.3. Les informations contenues dans la requète

De nombreuses informations en provenance du client peuvent être extraites de l'objetServletRequest passé en paramètre par le serveur (ou de HttpServletRequest qui hérite de ServletRequest).

Les informations les plus utiles sont les paramètres envoyés dans la requète.

L'interface ServletRequest dispose de nombreuses méthodes pour obtenir ces informations :

Méthode	Role
int getContentLength()	Renvoie la taille de la requète, 0 si elle est inconnue
String getContentType()	Renvoie le type MIME de la requète, null si il est inconnu
ServletInputStream getInputStream()	Renvoie un flux qui contient le corps de la requète
Enumeration getParameterNames()	Renvoie une énumeration contenant le nom de tous les paramètres
String getProtocol()	Retourne le nom du protocol et sa version utilisé par la requète
BufferedReader getReader()	Renvoie un flux qui contient le corps de la requète
String getRemoteAddr()	Renvoie l'adresse IP du client
String getRemoteHost()	Renvoie le nom de la machine cliente
String getScheme	Renvoie le protocole utilisé par la requète (exemple : http, ftp ...)
String getServerName()	Renvoie le nom du serveur qui à reçu la requète
int getServerPort()	Renvoie le port du serveur qui a reçu la requète

Exemple (code java 1.1) :

```
package tomcat1;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class InfoServlet extendsHttpServlet {

   public voiddoGet(HttpServletRequest request, HttpServletResponse response)
   throws IOException,ServletException {

      GenererReponse(request, response);

   }

   protected voidGenererReponse(HttpServletRequest request, HttpServletResponse reponse)
   throws IOException {

      reponse.setContentType("text/html");

      PrintWriter out =reponse.getWriter();

      out.println("<html>");
      out.println("<body>");
      out.println("<head>");
      out.println("<title>Informationsa disposition de la servlet</title>");
      out.println("</head>");
      out.println("<body>");
```

```
      out.println("<p>Typemime de la requète :"
        +request.getContentType()+"</p>");
      out.println("<p>Protocolede la requète :"
        +request.getProtocol()+"</p>");
      out.println("<p>AdresseIP du client :"
        +request.getRemoteAddr()+"</p>");
      out.println("<p>Nom duclient : "
        +request.getRemoteHost()+"</p>");
      out.println("<p>Nom duserveur qui a reçu la requète :"
        +request.getServerName()+"</p>");
      out.println("<p>Port duserveur qui a reçu la requète :"
        +request.getServerPort()+"</p>");
      out.println("<p>scheme: "+request.getScheme()+"</p>");
      out.println("<p>listedes parametres </p>");

      for (Enumeration e =request.getParameterNames() ; e.hasMoreElements() ; ) {

         Object p = e.nextElement();
         out.println("<p>  nom : "+p+" valeur :"
            +request.getParameter(""+p)+"</p>");

      }

      out.println("</body>");
      out.println("</html>");

   }
}
```

Résultat : avec l'url http://localhost:8080/examples/servlet/tomcat1.InfoServlet?param1=valeur1&param2=valeur2 : Une page html s'affiche contenant :

```
Type mime de la requète : null

Protocole de la requète : HTTP/1.0

Adresse IP du client : 127.0.0.1

Nom du client : localhost

Nom du serveur qui a reçu la requète : localhost

Port du serveur qui a reçu la requète : 8080

scheme : http

liste des parametres

   nom : param2 valeur :valeur2

   nom : param1 valeur :valeur1
```

26.6. La mise en oeuvre des servlets

26.6.1. Jakarta tomcat

26.6.1.1. L'installation sur Windows 9x

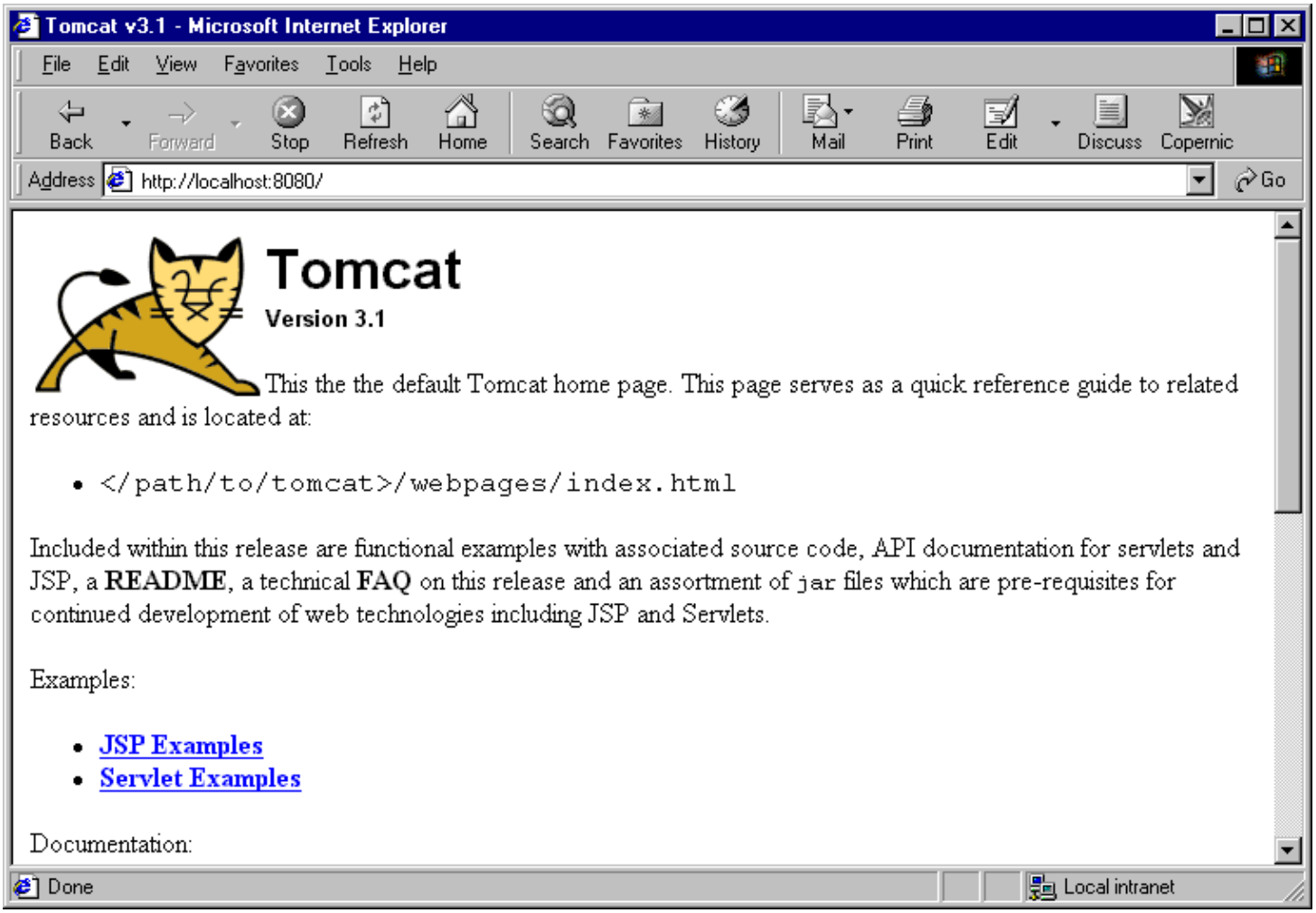
Il est possible de récuperer tomcat sur le site de Jakarta. Il faut choisir la version stable de préférence et le répertoire bin pour récuperer le fichier jakarta−tomcat.zip

Dézipper le fichier par exemple dans C:\. L'archive est décompressée dans un répertoire nommé jakarta−tomcat

Dans une boite DOS, assigner le répertoire contenant tomcat dans une variable d'environnement TOMCAT_HOME
Exemple : set TOMCAT_HOME=c:\jakarta−tomcat

Lancer Tomcat en executant le fichier startup.bat dans le répertoire TOMCAT_HOME\bin

Vérifier que Tomcat s'execute en saisissant l'url http://localhost:8080 dans un browser. La page d'accueil de Tomcat s'affiche.

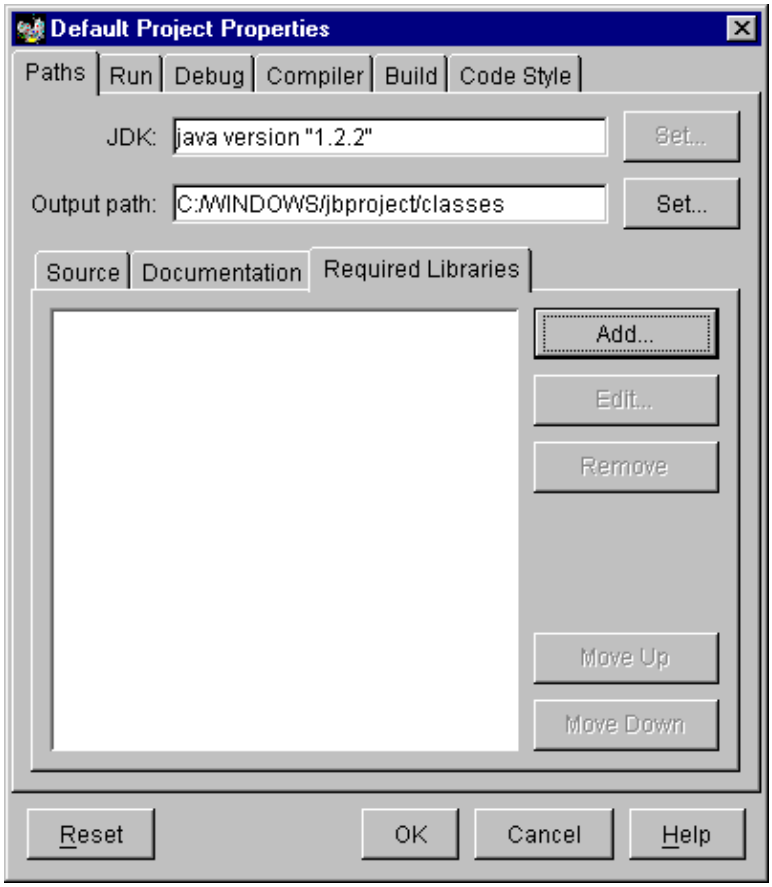


Le script %TOMCAT_HOME%\bin\shutdown.bat permet de stopper Tomcat.

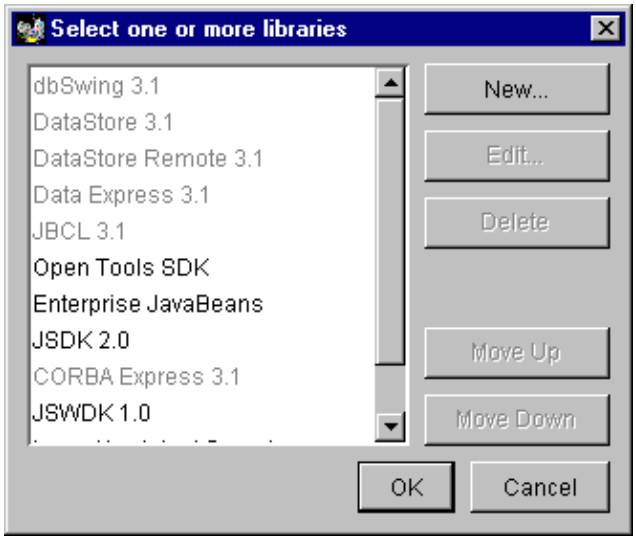
26.6.1.2. L'utilisation avec Jbuilder 3.0

26.6.1.2.1. La définition d'une nouvelle bibliothèque

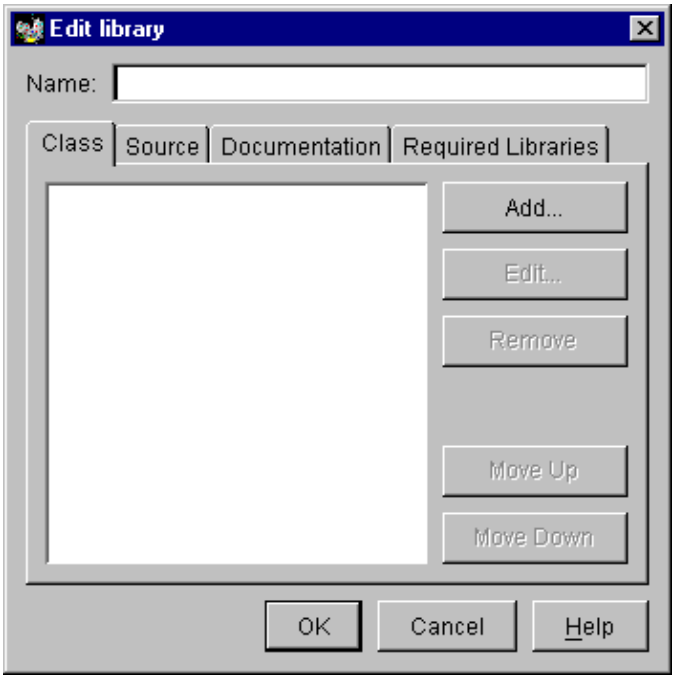
Pour définir une nouvelle bibliothèque, il faut sélectionner l'option default project properties dans le menu project



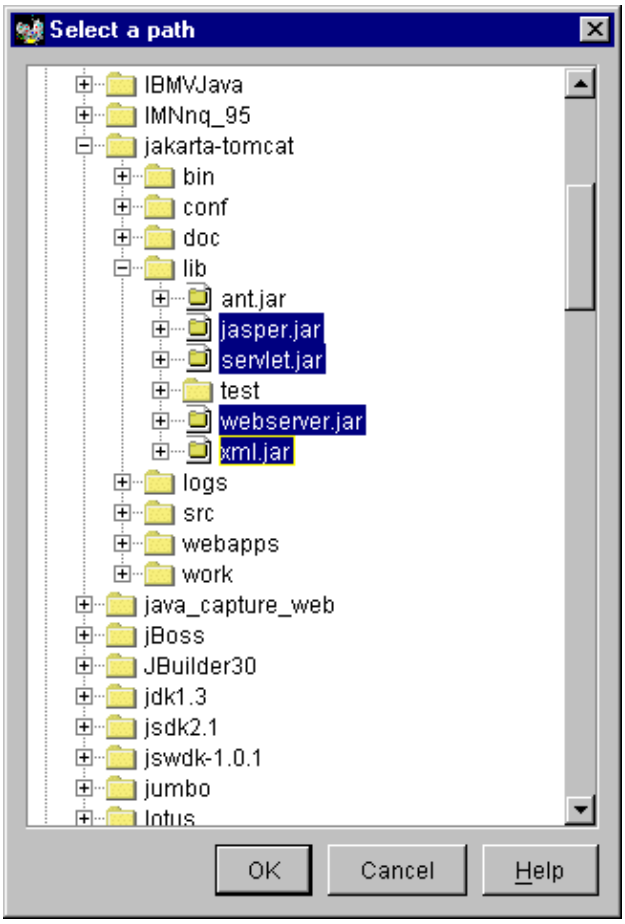
dans l'onglet required properties, cliquer sur le bouton " add "



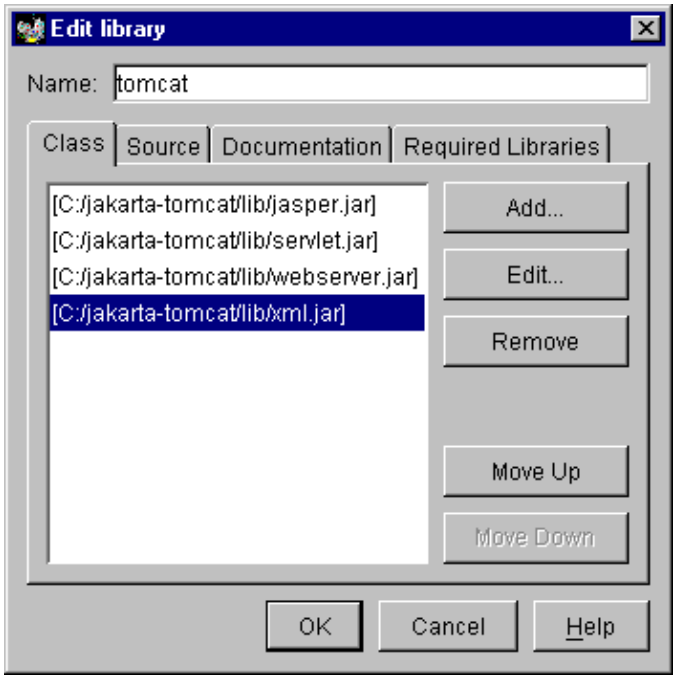
cliquer sur le bouton " new "



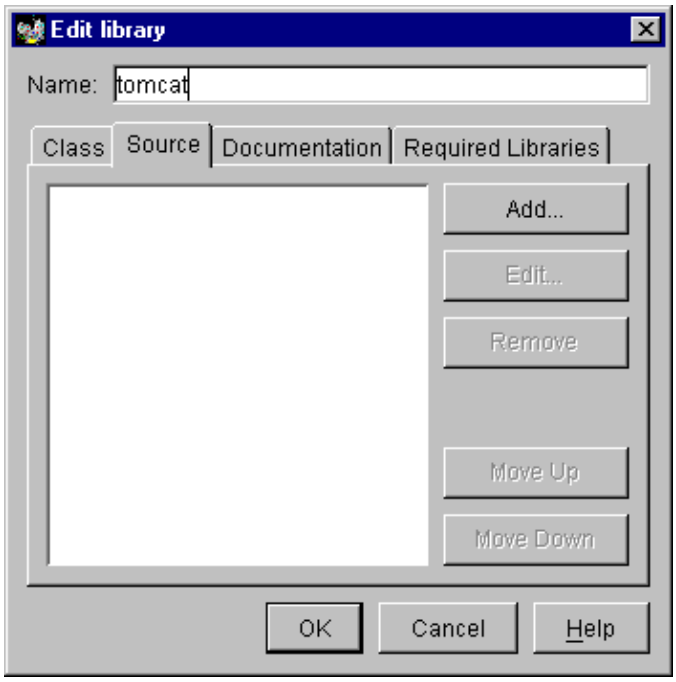
Saisir un nom, par exemple tomcat. Dans l'onglet classe, cliquer sur le bouton " add "



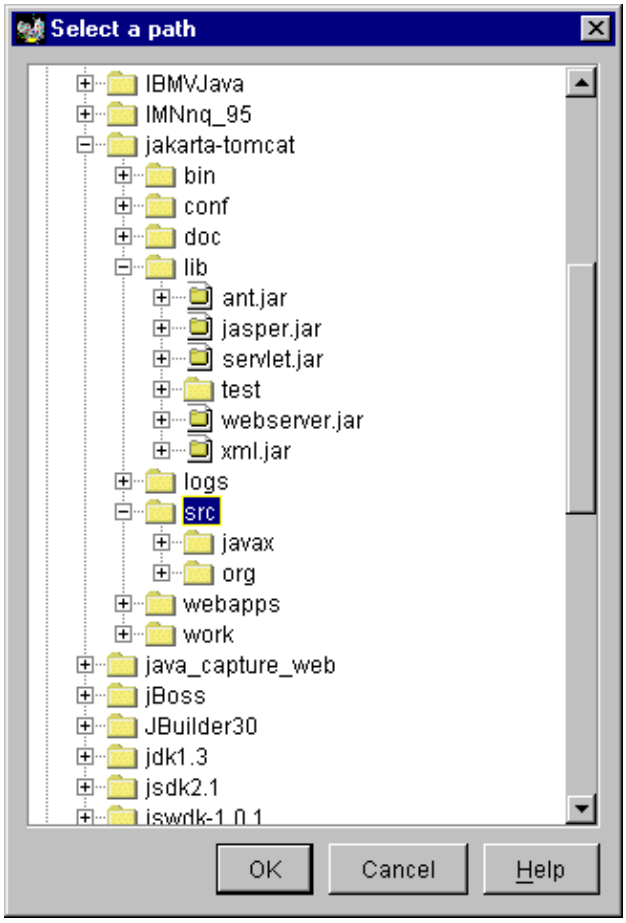
Sélectionner les fichiers jasper.jar, servlet.jar, webserver.jar et xml.jar dans le répertoire %TOMCAT_HOME%\lib et cliquer sur le bouton " ok "



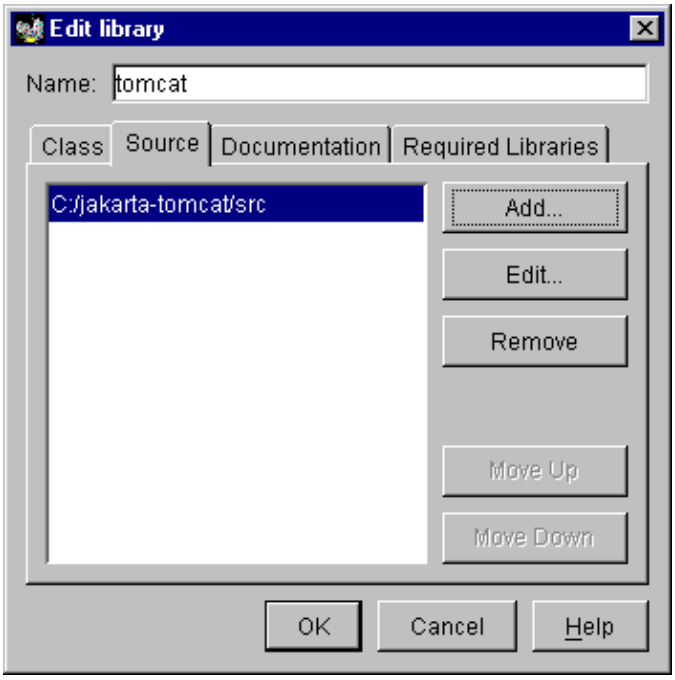
Selectionner l'onglet source



Cliquer sur le bouton " add "



Sélectionner le répertoire %TOMCAT_HOME%\src et cliquer le sur le bouton " ok "



Cliquer sur le bouton " ok " pour fermer la boîte de dialogue et revenir à la précédente.

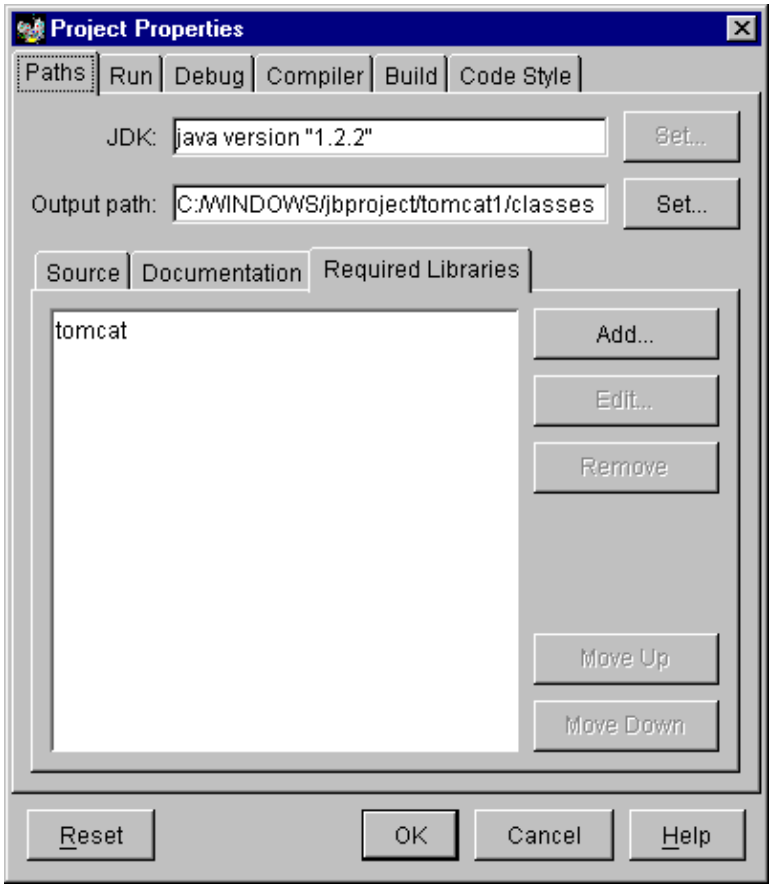
Cliquer sur le bouton " cancel " 2 fois (sur la boite de dialogue courante et la suivante) pour ne pas modifier les paramètres par défaut des projets.

26.6.1.2.2. La creation d'un nouveau projet

Créer un nouveau projet en sélectionnant l'option New project du menu file. Nommer ce projet par exemple tomcat1.

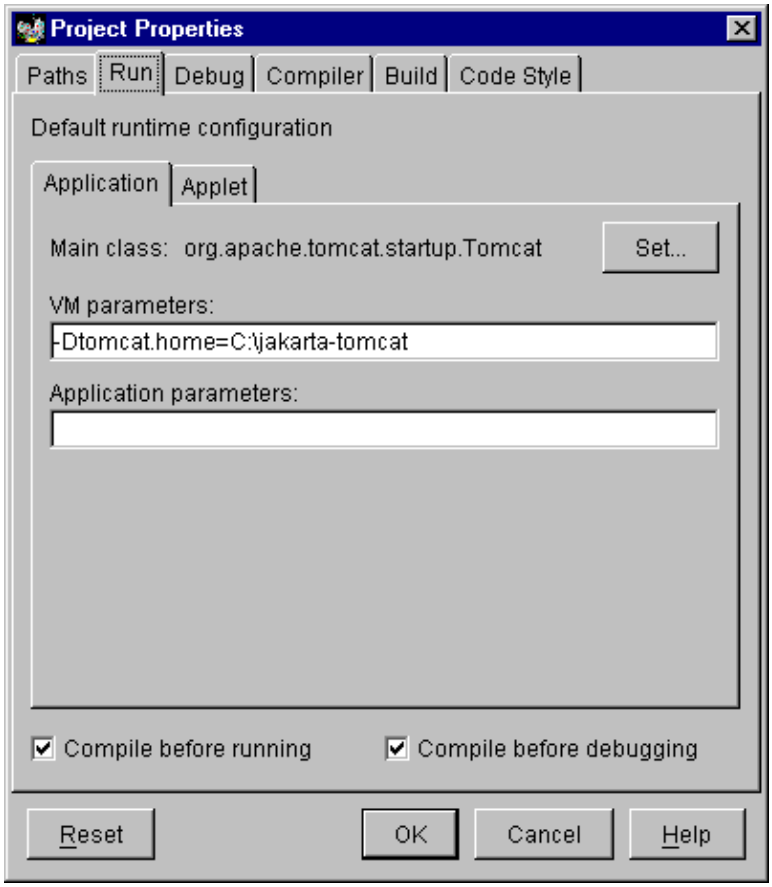
Sélectionner l'option Menu project/project properties

Sur l'onglet Paths, dans l'onglet required librairies, appuyer sur add et sélectionner la librairie tomcat précédemment définie.

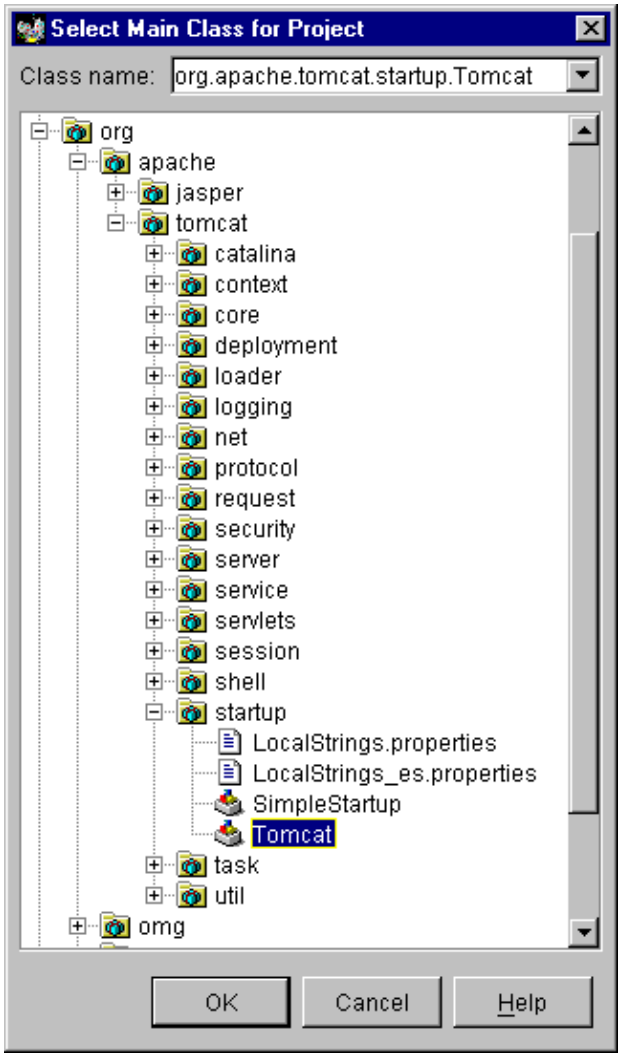


Activer l'onglet Run

Ajouter l'option −Dtomcat.home=C:\jakarta−tomcat dans la zone de saisie VM parameters.



Mettre la classe org.apache.tomcat.startup.tomcat comme classe principale du projet en appuyant sur le bouton " Set .. " et en choisissant la classe dans l'arborescence.



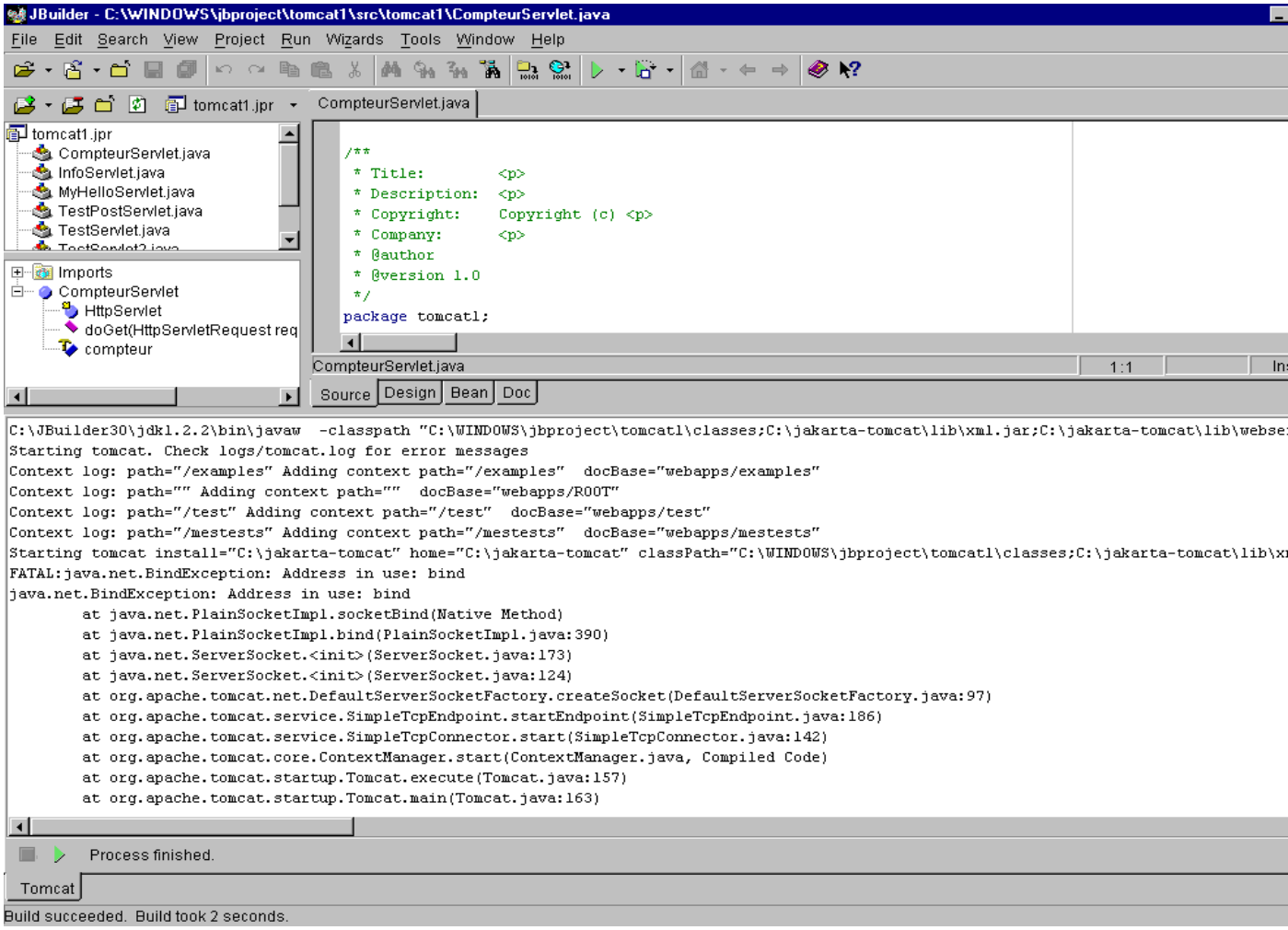
Si les classes n'apparaisent pas, fermer et rouvrir la boite de dialoque " project properties "

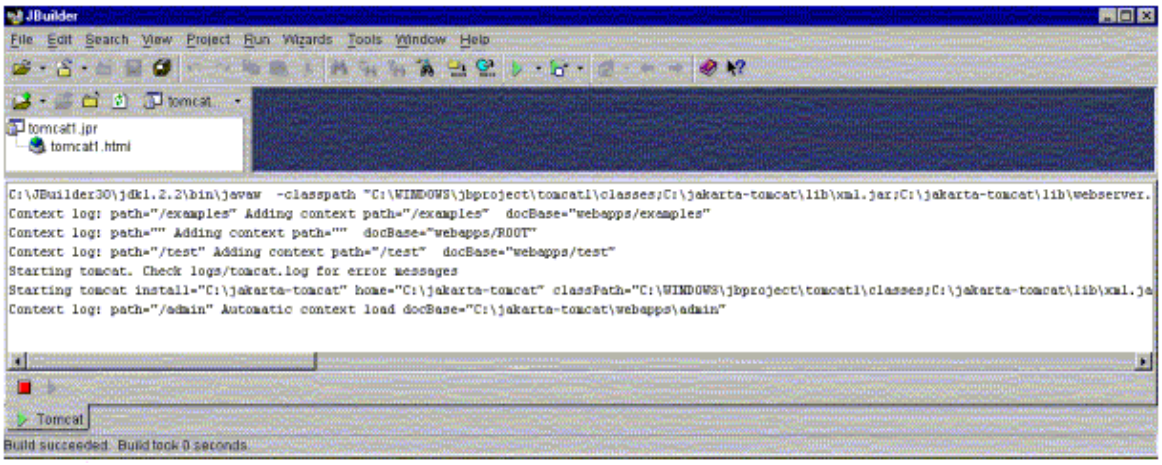
Cliquer sur " ok "

Fermer la boite en cliquant sur " ok ".

Remarque : ce paramétrage sera à faire pour tous les nouveaux projets qui contiendront des servlets executées avec Tomcat.

Executer le projet (menu run/run project ou F9)

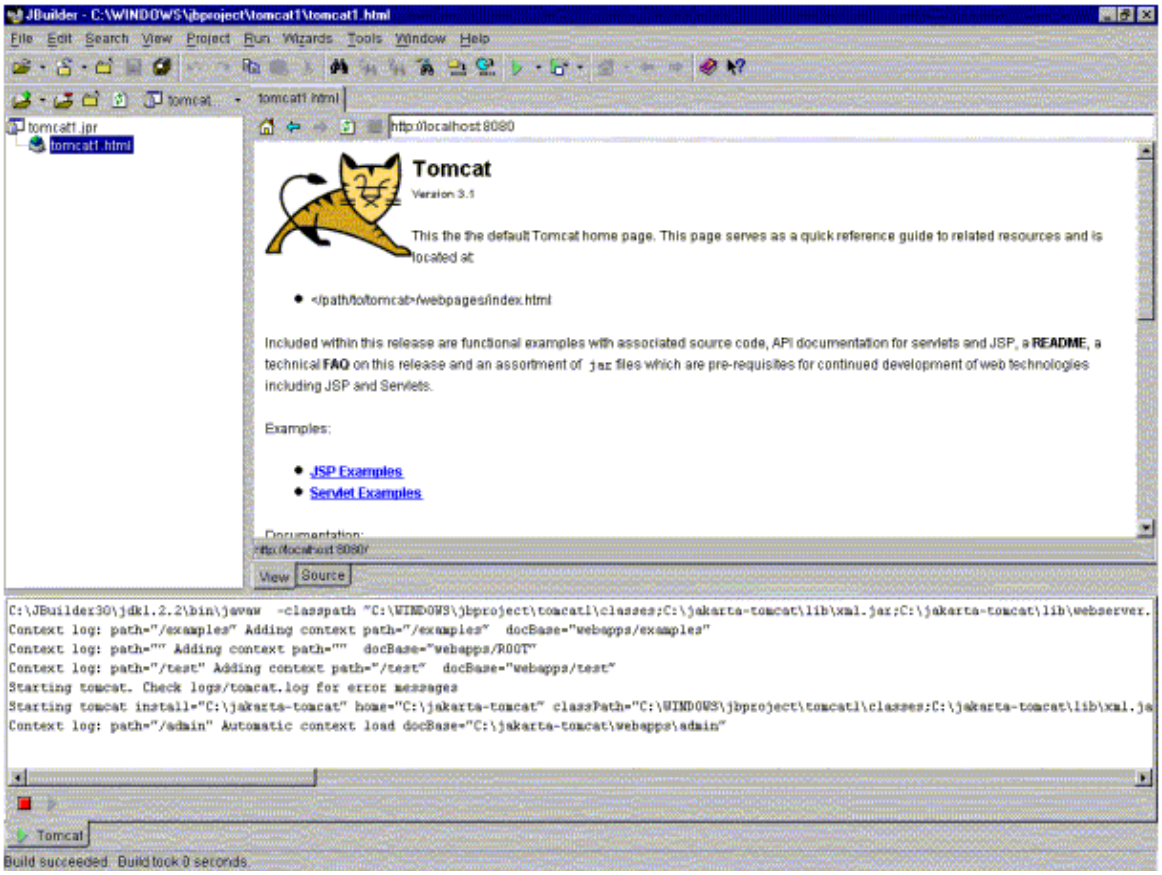




Il est possible de vérifier le bon fonctionnement en le visualisant dans le browser.

Double clic sur tomcat1.html.

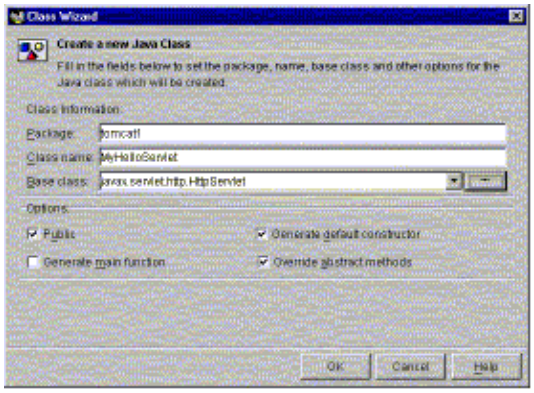
Dans la zone de saisie de l'url, saisie http://localhost:8080



Un clic sur le petit carré rouge au dessus de l'onglet " tomcat " permet d'arreter tomcat. Le triangle vert permet de le lancer.

26.6.1.2.3. La creation d'une servlet

Sélectionner l'option " new class " du menu " file " pour ajouter une classe.



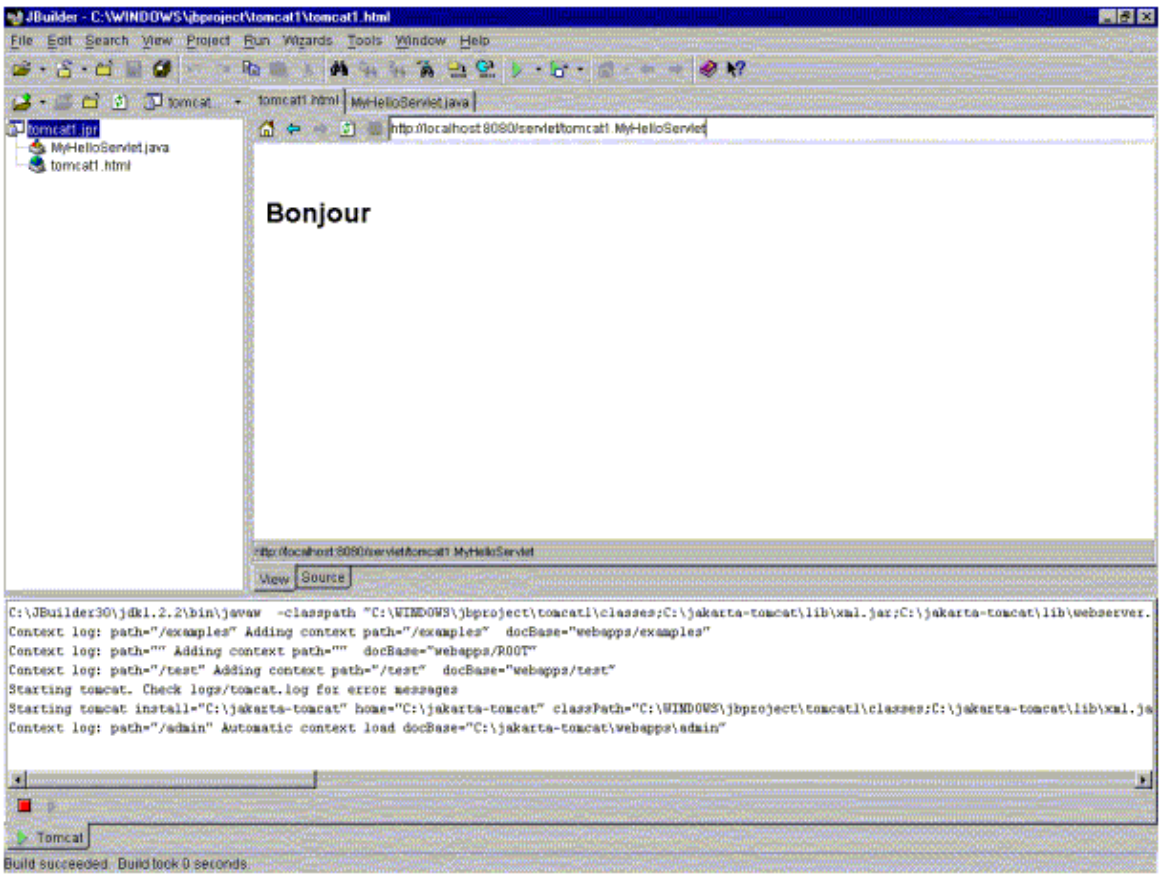
Exemple (code java 1.1) : saisir le code de la classe

```
package tomcat1;

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

public class MyHelloServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
```

```
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");
        out.println("<title>Bonjour</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Bonjour</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

26.6.1.2.4. Le test de la servlet

Pour tester : le projet doit etre en cours d'execution (onglet tomcat avec un triangle vert) Saisir L'url : http://localhost:8080/servlet/tomcat1.MyHelloServlet



Pour tester les modifications faites dans les servlets, il faut arreter le serveur tomcat et le relancer (clic sur le carré rouge puis sur le triangle vert)

26.7. L'utilisation des cookies

Les cookies sont des fichiers contenant des données au format texte, envoyés par le serveur et stocké sur le poste client. Les données contenues dans le cookie sont envoyées au serveur à chaque requête.

Les cookies peuvent être utilisés explicitement ou implicitement par exemple lors de l'utilisation d'une session.

Les cookies ne sont pas dangereux car ce sont uniquement des fichiers textes qui ne sont pas executés. De plus les navigateurs posent des limites sur le nombre (en principe 20 cookie pour un même serveur) et la taille des cookies (4ko maximum). Par contre les cookies peuvent contenir des données plus ou moins sensibles.

26.7.1. La classe Cookie

La classe javax.servlet.http.Cookie encapsule un cookie.

Un cookie est composé d'un nom, d'une valeur et d'attributs.

Pour créer un cookie, il suffit d'instancier un nouvel objet Cookie. La classe Cookie ne possède qu'un seul conctructeur qui attend deux paramètres de type String : le nom et la valeur associée.

Le classe Cookie possède plusieurs getter et setter pour obtenir ou définir des attributs qui sont tous optionnels.

Attribut	Rôle
Comment	Commentaire associé au cookie
Domain	Le nom de domaine (partiel ou complet) associé au cookie. Seul les serveurs contenant se nom de domaine recevront le cookie.
MaxAge	Durée de vie en seconde du cookie. Une fois ce délai expiré, le cookie est détruit sur le poste client. Une valeur actuelle, limite la durée de vie du cookie à la durée de vie du browser
Name	Nom du cookie
Path	Chemin du cookie. Ce chemin permet de renvoyer le cookie uniquement au serveur dont l'url contient également le chemin. Par défaut, cet attribut contient le chemin de l'url de la servlet. Par exemple, pour que le cookie soit renvoyé à toutes les requêtes du serveur, il suffit d'affecter la valeur "/" à cette attribut.
Secure	Booléen qui précise si le cookie ne doit être envoyé que via une connexion SSL.
Value	Valeur associée au cookie.
Version	Version du protocole utilisé pour gérer le cookie

26.7.2. L'enregistrement et la lecture d'un cookie

Pour envoyer un cookie au browser, il suffit d'utiliser la méthode addCookie() de la classe HttpServletResponse.

Exemple :

```
Cookie monCookie = new Cookie("nom","valeur");
response.addCookie(monCookie);
```

Pour lire un cookie envoyé par le browser, il faut utiliser la méthode getCookies() de la classe HttpServletRequest. Cette méthode renvoie un tableau d'objet Cookie. Les cookies sont renvoyés dans l'en−tête de la requête http. Pour rechercher un cookie particulier, il faut parcourir le tableau et rechercher le cookie à partir de son nom grace à la méthode getName() de l'objet Cookie.

Exemple :

```
Cookie[] cookies = request.getCookies();
String valeur = "";
for(int i=0;i<cookies.length;i++) {
```

```
  if(cookies[i].getName().equals("nom")) {
    valeur=cookies[i].getValue();
  }
}
```

26.8. Le partage d'informations entre plusieurs échanges HTTP

Les sections suivantes sont en cours d'écriture

27. Les JSP (Java Servers Pages)

Les JSP (Java Server Pages) sont une technologie basée sur Java qui permettent la génération de pages web dynamiques.

La technologie JSP permet de séparer la présentation sous forme de code HTML et les traitements sous formes de classes java définissant un bean ou une servlet. Ceci est d'autant plus facile que les JSP définissent une syntaxe particulière permettant d'appeler un bean et d'insérer le résultat de son traitement dans la page HTML dynamiquement.

Les informations fournies dans ce chapitre concerne uniquement les spécifications 1.0 et 1.1 des JSP.

27.1. Présentation des JSP

Les JSP permettent d'introduire du code java dans des tags prédéfinis à l'intérieur d'une page HTML. La technologie JSP mélange la puissance de java côté serveur et la facilité de mise en page d'HTML côté client.

Sun fourni de nombreuses informations sur la technologie JSP à l'adresse suivante : http://java.sun.com/products/jsp/index.html

Une JSP est habituellement constituée :

- de données et de tags HTML
- de tags JSP
- de scriptlets (code java intégré à la JSP)

Les fichiers JSP possèdent par convention l'extension .jsp.

Concrètement, les JSP sont basées sur les servlets. Au premier appel de la page JSP, le moteur de JSP crée et compile automatiquement une servlet qui permet la génération de la page web. Le code HTML est repris intégralement dans la servlet. Le code java est inséré dans la servlet.

La servlet est sauvegardée puis elle est exécutée. Les appels suivants de la JSP sont beaucoup plus rapides car la servlet, conservée par le serveur, est directement exécutée.

Il a plusieurs manières de combiner les technologies JSP, les beans/EJB et les servlets.

Comme le code de la servlet est généré dynamiquement, les JSP sont relativement difficiles à debugguer.

Cette approche possède plusieurs avantages :

- l'utilisation de java par les JSP permet une indépendance de la plate−forme d'exécution mais aussi du serveur web utilisé.
- la séparation des traitements et de la présentation : la page web peut être écrite par un designer et les tags java peuvent être ajouter ensuite par le développeur. Les traitements peuvent être réalisés par des composants réutilisables (des java beans).
- les JSP sont basées sur les servlets : tous ce qui est fait par une servlet pour la génération de pages dynamiques peut être fait avec une JSP.

27.1.1. Le choix entre JSP et Servlets

Les servlets et les JSP ont de nombreux points communs puisque qu'une JSP est finalement convertie en une servlet. Le choix d'utiliser l'une ou l'autre de ces technologies ou les deux doit être fait pour tirer le meilleur parti de leurs avantages.

Dans une servlet, les traitements et la présentation sont regroupés. L'aspect présentation est dans ce cas pénible à développer et à maintenir à cause de l'utilisation répétitive de méthodes pour insérer le code HTML dans le flux de sortie. De plus, une simple petite modification dans le code HTML nécessite la recompilation de la servlet. Avec un JSP, la séparation des traitements et de la présentation rend ceci très facile et automatique.

Il est préférable d'utiliser les JSP pour générer des pages web dynamiques.

L'usage des servlets est obligatoire si celles ci doivent communiquer directement avec une applets ou une application et non plus avec un serveur web.

27.1.2. JSP et les technologies concurrentes

Il existe plusieurs technologies similaires aux JSP notamment ASP et PHP.

27.2. Les outils nécessaires

Dans un premier temps, Sun a fourni un kit de développement pour les JSP : le Java Server Web Development Kit (JSWDK). Actuellement, Sun a chargé le projet Apache de développer l'implémentation officielle d'un moteur de JSP. Ce projet se nomme Tomcat.

En fonction des versions des API utilisées, il faut choisir un produit différent. Le tableau ci dessous résume le produit a utiliser en fonction de la version des API utilisée.

Produit	Version	Version de l'API servlet implémentée	Version de l'API JSP implémentée
JSWDK	1.0.1	2.1	1.0
Tomcat	3.2	2.2	1.1
Tomcat	4.0	2.3	1.2

Ces produits sont librement téléchargeables sur le site de Sun à l'adresse suivante : http://java.sun.com/products/jsp/download.html

Pour télécharger le JSWDK, il faut cliquer sur le lien « archive ».

27.2.1. JavaServer Web Development Kit (JSWDK) sous Windows

Le JSWDK est sous la forme d'un fichier zip nommé jswdk_1_0_1−win.zip.

Pour l'installer, il suffit de décompacter l'archive.

Pour lancer le serveur :

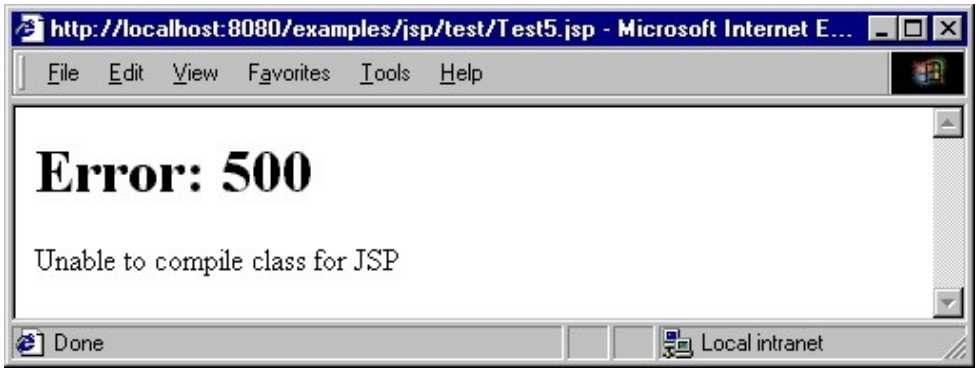
```
C:\jswdk-1.0.1>startserver.bat

Using classpath:.\classes;.\webserver.jar;.\lib\jakarta.jar;.\lib\servlet.jar;.
\lib\jsp.jar;.\lib\jspengine.jar;.\examples\WEB-INF\jsp\beans;.\webpages\WEB-INF
\servlets;.\webpages\WEB-INF\jsp\beans;.\lib\xml.jar;.\lib\moo.jar;\lib\tools.ja
r;C:\jdk1.3\lib\tools.jar;
C:\jswdk-1.0.1>
```

Le serveur s'exécute dans une console en tache de fond. Cette console permet de voir les messages émis par le serveur.

Exemple : au démarrage

```
JSWDK WebServer Version 1.0.1
Loaded configuration from: file:C:\jswdk-1.0.1\webserver.xml
endpoint created: localhost/127.0.0.1:8080
```

Si la JSP contient une erreur, le serveur envoie une page d'erreur :



Une exception est levée et est affichée dans la fenêtre ou le serveur s'exécute :

Exemple :

```
-- Commentaires de la page JSP --
^
1 error
at com.sun.jsp.compiler.Main.compile(Main.java:347)
at com.sun.jsp.runtime.JspLoader.loadJSP(JspLoader.java:135)
at com.sun.jsp.runtime.JspServlet$JspServletWrapper.loadIfNecessary(JspS
ervlet.java:77)
at com.sun.jsp.runtime.JspServlet$JspServletWrapper.service(JspServlet.j
ava:87)
at com.sun.jsp.runtime.JspServlet.serviceJspFile(JspServlet.java:218)
at com.sun.jsp.runtime.JspServlet.service(JspServlet.java:294)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:840)
at com.sun.web.core.ServletWrapper.handleRequest(ServletWrapper.java:155
)
at com.sun.web.core.Context.handleRequest(Context.java:414)
at com.sun.web.server.ConnectionHandler.run(ConnectionHandler.java:139)
HANDLER THREAD PROBLEM: java.io.IOException: Socket Closed
java.io.IOException: Socket Closed
at java.net.PlainSocketImpl.getInputStream(Unknown Source)
at java.net.Socket$1.run(Unknown Source)
at java.security.AccessController.doPrivileged(Native Method)
at java.net.Socket.getInputStream(Unknown Source)
at com.sun.web.server.ConnectionHandler.run(ConnectionHandler.java:161)
```

Le répertoire work contient le code et le byte code des servlets générée à partir des JSP.

Pour arrêter le serveur, il suffit d'exécuter le script stopserver.bat.

A l'arrêt du serveur, le répertoire work qui contient les servlets générées à partir des JSP est supprimé.

27.2.2. Tomcat

Cette section est en cours d'écriture

27.3. Le code HTML

Une grande partie du contenu d'une JSP est constitué de code HTML. D'ailleurs, le plus simple pour écrire une JSP est d'écrire le fichier HTML avec un outils dédié et d'ajouter ensuite les tags JSP pour ce qui concerne les parties dynamiques.

La seule restriction concernant le code HTML concerne l'utilisation dans la page générée du texte « <% » et « %> ». Dans ce cas, le plus simple est d'utiliser les caractères spéciaux HTML &lt ; et &gt ;. Sinon l'analyseur syntaxique du moteur de JSP considère que c'est un tag JSP et renvoie une erreur.

Exemple :

```
<HTML>
<HEAD>
<TITLE>Test</TITLE>
</HEAD>
<BODY>
<p>Plusieurs tags JSP commencent par &lt;% et se finissent par %&gt;</p>
</BODY>
</HTML>
```

27.4. Les Tags JSP

Il existe trois types de tags :

- tags de directives : ils permettent de contrôler la structure de la servlet générée
- tags de scripting: il permettent d'insérer du code java dans la servlet
- tags d'actions: ils facilitent l'utilisation de composants

Attention : Les tags sont sensibles à la casse.

27.4.1. Les tags de directives <%@ … %>

Les spécifications des JSP définissent trois directives :

- page : permet de définir des options de configuration
- include : permet d'inclure des fichiers statiques dans la JSP avant la génération de la servlet
- taglib : permet de définir des tags personnalisés

Leur syntaxe est la suivante :

<%@ directive attribut="valeur" ... %>

27.4.1.1. La directive page

Cette directive doit être utilisée dans toutes les pages JSP : elle permet de définir des options qui s'applique à toute la JSP.

Elle peut être placée n'importe ou dans le source mais il est préférable de la mettre en début de fichier, avant même le tag <HTML>. Elle peut être utilisée plusieurs fois dans une même page mais elle ne doit définir la valeur d'une option qu'une seule fois, sauf pour l'option import.

Les options définies par cette directive sont de la forme option=valeur.

Option	Valeur	Valeur par défaut	Autre valeur possible
autoFlush	Une chaine	«true»	«false»
buffer	Une chaine	«8kb»	«none» ou «nnnkb» (nnn indiquant la valeur)
contentType	Une chaîne contenant le type mime		
errorPage	Une chaîne contenant une URL		
extends	Une classe		
import	Une classe ou un package.*		
info	Une chaîne		
isErrorPage	Une chaîne	«false»	«true»
isThreadSafe	Une chaîne	«true»	«false»
langage	Une chaîne	«java»	
session	Une chaîne	«true»	«false»

Exemple :

```
<%@ page import="java.util.*" %>
<%@ page import="java.util.Vector" %>
<%@page info="Ma premiere JSP"%>
```

Les options sont :

- autoFlush="**true**|false"

  Cette option indique si le flux en sortie de la servlet doit être vidé quand le tampon est plein. Si la valeur est false, une exception est levée dès que le tampon est plein. On ne peut pas mettre false si la valeur de buffer est none.

- buffer="none|**8kb**|*size*kb"

  Cette option permet de préciser la taille du buffer des données générées contenues par l'objet out de type JspWriter.

- contentType="*mimeType* [; charset=*characterSet*]" | "**text/html;charset=ISO−8859−1**"

  Cette option permet de préciser le type MIME des données générées.

  Cette option est équivalente à <% response.setContentType("mimeType"); %>

- errorPage=*"relativeURL"*

  Cette option permet de préciser la JSP appelée au cas où une exception est levée

  Si l'URL commence pas un ‘/', alors l'URL est relative au répertoire principale du serveur web sinon elle est relative au répertoire qui contient la JSP

- extends=*"package.class"*

  Cette option permet de préciser la classe qui sera la super classe de l'objet java créé à partir de la JSP.

- import= "{ *package.class* | *package.** }, ..."

  Cette option permet d'importer des classes contenues dans des packages utilisées dans le code de la JSP. Cette option s'utilise comme l'instruction import dans un source java.

  Chaque classe ou package est séparée par une virgule.

  Cette option peut être présente dans plusieurs directives page.

- info="text"

  Cette option permet de préciser un petit descriptif de la JSP. Le texte fourni sera renvoyé par la méthode getServletInfo() de la servlet générée.

- isErrorPage="true|**false**"

  Cette option permet de préciser si la JSP génère un page d'erreur. La valeur true permet d'utiliser l'objet Exception dans la JSP

- isThreadSafe="**true**|false"

  Cette option indique si la servlet générée sera multithread : dans ce cas, une même instance de la servlet peut gérer plusieurs requêtes simultanément. En contre partie, elle doit gérer correctement les accès concurrents aux ressources. La valeur false impose à la servlet générée d'implémenter l'interface SingleThreadModel.

- language="java"

  Cette option définie le langage utilisé pour écrire le code dans la JSP. La seule valeur autorisée actuellement est «java».

- session="**true**|false**"**

  Cette option permet de préciser si la JSP est incluse dans une session ou non. La valeur par défaut (true) permet l'utilisation d'un objet session de type HttpSession qui permet de gérer une session.

27.4.1.2. La directive include

Cette directive permet d'inclure un fichier dans le source JSP. Le fichier inclus peut être un fragment de code JSP, HTML ou java. Le fichier est inclus dans la JSP avant que celle ci ne soit interprétée par le moteur de JSP.

Ce tag est particulièrement utile pour insérer un élément commun à plusieurs pages tel qu'un en−tête ou un bas de page.

Si le fichier inclus est un fichier HTML, celui ci ne doit pas contenir de tag <HTML>, </HTML>, <BODY> ou </BODY> qui ferait double emploi avec ceux présents dans le fichier JSP. Ceci impose d'écrire des fichiers HTML particuliers uniquement pour être inclus dans les JSP : ils ne pourront pas être utilisés seuls.

La syntaxe est la suivante :

<%@ include file="chemin relatif du fichier" %>

Si le chemin commence par un ‘/', alors le chemin est relatif au contexte de l'application, sinon il est relatif au fichier JSP.

Exemple :

```
bonjour.htm :

<p><table border="1" cellpadding="4" cellspacing="0" width="30%" align=center >
<tr bgcolor="#A6A5C2">
<td align="center">BONJOUR</Td>
</Tr>
</table></p>

Test1.jsp :

<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<p align="center">Test d'inclusion d'un fichier dans la JSP</p>
<%@ include file="bonjour.htm"%>
<p align="center">fin</p>
</BODY>
</HTML>
```

Pour tester cette JSP avec le JSWDK, il suffit de placer ces deux fichiers dans le répertoire jswdk−1.0.1\examples\jsp\test.

Pour visualiser la JSP, il faut saisir l'url http://localhost:8080/examples/jsp/test/Test1.jsp dans un navigateur.

Attention : un changement dans le fichier inclus ne provoque pas une regénération et une compilation de la servlet correspondant à la JSP. Pour insérer un fichier dynamiquement à l'exécution de la servlet il faut utiliser le tag <jsp:include>.

27.4.1.3. La directive taglib

Cette section est en cours d'écriture

27.4.2. Les tags de scripting

Ces tags permettent d'insérer du code java qui sera inclus dans la servlet générée à partir de la JSP. Il existe trois tags pour insérer du code java :

- le tag de déclaration : le code java est inclus dans le corps de la servlet générée. Ce code peut être la déclaration de variables d'instances ou de classes ou la déclaration de méthodes.
- le tag d'expression : évalue une expression et insère le résultat sous forme de chaîne de caractères dans la page web générée.
- le tag de scriptlets : par défaut, le code java est inclus dans la méthode service() de la servlet.

Il est possible d'utiliser dans ces tags plusieurs objets définis par les JSP.

27.4.2.1. Le tag de déclarations <%! … %>

Ce tag permet de déclarer des variables ou des méthodes qui pourront être utilisées dans la JSP. Il ne génère aucun caractère dans le fichier HTML de sortie.

La syntaxe est la suivante :

<%! declarations %>

Exemple :

```
<%! int i = 0; %>
<%! dateDuJour = new java.util.Date(); %>
```

Les variables ainsi déclarées peuvent être utilisées dans les tags d'expressions et de scriptlets.

Il est possible de déclarer plusieurs variables dans le même tag en les séparant avec des ‘ ;'.

Ce tag permet aussi d'insérer des méthodes dans le corps de la servlet.

Exemple :

```
<HTML>
<HEAD>
<TITLE>Test</TITLE>
</HEAD>
<BODY>
<%!
int minimum(int val1, int val2) {
   if (val1 < val2) return val1;
   else return val2;
}
%>
<% int petit = minimum(5,3);%>
<p>Le plus petit de 5 et 3 est <%= petit %></p>
</BODY>
</HTML>
```

27.4.2.2. Le tag d'expressions <%= … %>

Le moteur de JSP remplace ce tag par le résultat de l'évaluation de l'expression présente dans le tag.

Ce résultat est toujours converti en une chaîne. Ce tag est un raccourci pour éviter de faire appel à la méthode println() lors de l'insertion de données dynamiques dans le fichier HTML.

La syntaxe est la suivante :

<%= expression %>

Le signe ‘=' doit être collé au signe ‘%'.

Attention : il ne faut pas mettre de ‘ ;' à la fin de l'expression.

Exemple : Insertion de la date dans la page HTML

```
<%@ page import="java.util.*" %>
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
```

```
<BODY>
<p align="center">Date du jour :
<%= new Date() %>
</p>
</BODY>
</HTML>
```

Résultat :

Date du jour : Thu Feb 15 11:15:24 CET 2001

L'expression est évaluée et convertie en chaîne avec un appel à la méthode toString(). Cette chaîne est insérée dans la page HTML en remplacement du tag. Il est ainsi possible que le résultat soit une partie ou la totalité d'un tag HTML ou même JSP.

Exemple :

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<%="<H1>" %>
Bonjour
<%="</H1>" %>
</BODY>
</HTML>
```

Résultat : code HTML généré

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<H1>
Bonjour
</H1>
</BODY>
</HTML>
```

27.4.2.3 Les variables implicites

Les spécifications des JSP définissent plusieurs objets utilisables dans le code dont les plus utiles sont :

Object	Classe	Role
out	javax.servlet.jsp.JspWriter	Flux en sortie de la page HTML générée
request	javax.servlet.http.HttpServletRequest	Contient la requête
response	javax.servlet.http.HttpServletRequest	Contient la réponse
session	javax.servlet.http.HttpSession	Gère la session

27.4.2.4. Le tag des scriptlets <% … %>

Ce tag contient du code java : un scriptlet.

La syntaxe est la suivante :

<% code java %>

Exemple :

```
<%@ page import="java.util.Date"%>
<html>
<body>
<%! Date dateDuJour; %>
<% dateDuJour = new Date();%>
Date du jour : <%= dateDuJour %><BR>
</body>
</html>
```

Par défaut, le code inclus dans le tag est inséré dans la méthode service de la servlet générée à partir de la JSP.

Ce tag ne peut pas contenir autre chose que du code java : il ne pas par exemple contenir de tags HTML ou JSP. Pour faire cela, il faut fermer le tag du scriptlet, mettre le tag HTML ou JSP puis de nouveau commencer un tag de scriptlet pour continuer le code.

Exemple :

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<% for (int i=0; i<10; i++) { %>
<%= i %> <br>
<% }%>
</BODY>
</HTML>
```

Résultat : la page HTML générée

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
0 <br>
1 <br>
2 <br>
3 <br>
4 <br>
5 <br>
6 <br>
7 <br>
8 <br>
9 <br>
</BODY>
</HTML>
```

27.4.3. Les tags de commentaires

Il existe deux types de commentaires avec les JSP :

- les commentaires visibles dans le code HTML

- les commentaires invisibles dans le code HTML

27.4.3.1. Les commentaires HTML <!−− … −−>

Ces commentaires sont ceux définis par format HTML. Ils sont intégralement reconduit dans le fichier HTML généré. Il est possible d'insérer, dans ce tag, un tag JSP de type expression

La syntaxe est la suivante :

<!−− commentaires [<%= expression %>] −−>

Exemple :

```
<%@ page import="java.util.*" %>
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<!−− Cette page a ete generee le <%= new Date() %> −−>
<p>Bonjour</p>
</BODY>
</HTML>
```

Résultat :

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<!−− Cette page a ete generee le Thu Feb 15 11:44:25 CET 2001 −−>
<p>Bonjour</p>
</BODY>
</HTML>
```

Le contenu d'une expression incluse dans des commentaires est dynamique : sa valeur peut changer à chaque génération de la page en fonction de son contenu.

27.4.3.2. Les commentaires cachés <%−− … −−%>

Les commentaires cachés sont utilisés pour documenter la page JSP. Leur contenu est ignoré par le moteur de JSP et ne sont donc pas reconduit dans les données HTML générées.

La syntaxe est la suivante :

<%−− commentaires −−%>

Exemple :

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<%−− Commentaires de la page JSP −−%>
<p>Bonjour</p>
</BODY>
</HTML>
```

Résultat :

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<p>Bonjour</p>
</BODY>
</HTML>
```

Ce tag peut être utile pour éviter l'exécution de code lors de la phase de débuggage.

27.4.4. Les tags d'actions

27.4.4.1. Le tag <jsp:useBean>

Le tag <jsp:useBean> permet de localiser une instance ou d'instancier un bean pour l'utiliser dans la JSP.

L'utilisation d'un bean dans une JSP est très pratique car il peut encapsuler des traitements complexes et être réutilisable par d'autre JSP ou composants. Le bean peut notamment assurer l'accès à une base de données. L'utilisation des beans permet de simplifier les traitements inclus dans la JSP.

Lors de l'instanciation d'un bean, on précise la porté du bean. Si le bean demandé est déjà instancié pour la porté précisée alors il n'y pas de nouvelle instance du bean qui est créée mais sa référence est simplement renvoyée : le tag <jsp:useBean> n'instancie pas obligatoirement un objet.

Ce tag ne permet de traiter des EJB.

La syntaxe est la suivante :

<jsp:useBean
id="*beanInstanceNam*e"
scope="**page**|request|session|application"
{ class="*package.class*" |
type="*package.class*" |
class="*package.class*" type="*package.class*" |
beanName="{*package.class* | <%= *expression* %>}" type="*package.class*"
}
{ /> |
> ...
</jsp:useBean>
}

L'attribut id permet de donner un nom à la variable qui va contenir la référence sur le bean.

L'attribut scope permet de définir la portée durant laquelle le bean est défini et utilisable. La valeurs de cette attribut détermine la manière dont le tag localise ou instancie le bean. Les valeurs possibles sont :

Valeur	Rôle
page	c'est la valeur par défaut. Le bean est utilisable dans toute la page JSP ainsi que dans les fichiers statiques inclus.
Request	Le bean est accessible durant la durée de vie de la requête. La méthode getAttribute de l'objet request permet d'obtenir une référence sur le bean.

session	le bean est utilisable par toutes les JSP qui appartiennent à la même session que la JSP qui a instancié le bean. Le bean est utilisable tout au long de la session par toutes les pages qui y participent. La JSP qui créé le bean doit avoir l'attribut session = « true » dans sa directive page.
application	le bean est utilisable par toutes les JSP qui appartiennent à la même application que la JSP qui a instancié le bean. Le bean n'est instancié que lors du rechargement de l'application.

L'attribut class permet d'indiquer la classe du bean

L'attribut type permet de préciser le type de la variable qui va contenir la référence du bean. La valeur indiquée doit obligatoirement être une super classe du bean ou une interface implémentée par le bean (directement ou par héritage)

L'attribut beanName permet d'instancier le bean grâce à la méthode instanciate() de la classe Beans.

Exemple :

```
<jsp:useBean id="monBean" scope="session" class="test.MonBean" />
```

Dans cet exemple, une instance de MonBean est instancié un seul est unique fois lors de la session. Dans la même session, l'appel du tag <jsp:useBean> avec le même bean et la même portée ne feront que renvoyer l'instance créée. Le bean est accessible durant toute la session.

Le tag <jsp:useBean> recherche si une instance du bean existe avec le nom et la portée précisée. Si elle n'existe pas, alors une instance est créée. Si il y a instanciation du bean, alors les tags <jsp:setProperty> inclus dans le tag sont utilisés pour initialiser les propriétés du bean sinon ils sont ignorés. Les tags inclus entre les tags <jsp:useBean> et </jsp:useBean> ne sont exécutés que si le bean est instancié.

Exemple :

```
<jsp:useBean id="monBean" scope="session" class="test.MonBean" >
<jsp:setProperty name="monBean" property="*" />
</jsp:useBean>
```

Cet exemple a le même effet que le précédent avec une initialisation des propriétés du bean lors de son instanciation avec les valeur des paramètres correspondants.

Exemple complet :

```
TestBean.jsp

<html>
<HEAD>
<TITLE>Essai d'instanciation d'un bean dans une JSP</TITLE>
</HEAD>
<body>
<p>Test d'utilisation d'un Bean dans une JSP </p>
<jsp:useBean id="personne" scope="request" class="test.Personne" />
<p>nom initial = <%=personne.getNom() %></p>
<%
personne.setNom("mon nom");
%>
<p>nom mise à jour = <%= personne.getNom() %></p>
</body>
</html>

Personne.java

package test;
public class Personne {
  private String nom;
  private String prenom;
```

```
  public Personne() {
    this.nom = "nom par defaut";
    this.prenom = "prenom par defaut";
  }

  public void setNom (String nom) {
    this.nom = nom;
  }

  public String getNom() {
    return (this.nom);
  }

  public void setPrenom (String prenom) {
    this.prenom = prenom;
  }

  public String getPrenom () {
    return (this.prenom);
  }
}
```

Selon le moteur de JSP utilisé, les fichiers du bean doivent être placé dans un répertoire particulier pour être accessible par la JSP.

Pour tester cette JSP avec Tomcat, il faut compiler le bean Personne dans le répertoire c:\jakarta−tomcat\webapps\examples\web−inf\classes\test et placer le fichier TestBean.jsp dans le répertoire c:\jakarta−tomcat\webapps\examples\jsp\test.

27.4.4.2. Le tag <jsp:setProperty >

Le tag <jsp:setProperty> permet de mettre à jour la valeur d'un ou plusieurs attributs d'un Bean. Le tag utilise le setter (méthode getXXX() ou XXX est le nom de la propriété) pour mettre à jour la valeur. Le bean doit exister grâce à un appel au tag <jsp:useBean>.

Il existe trois façon de mettre à jour les propriétés soit à partir des paramètres de la requête soit avec une valeur :

- alimenter automatiquement toutes les propriétés avec les paramètres correspondants de la requête
- alimenter automatiquement une propriété avec le paramètre de la requête correspondant
- alimenter une propriété avec la valeur précisée

La syntaxe est la suivante :

<jsp:setProperty name="*beanInstanceName*"
{ property="*" |
property="*propertyName*" [param=" *parameterName*"] |
property="*propertyName*" value="{*string* | <%= *expression*%>}"

}
/>

L'attribut name doit contenir le nom de la variable qui contient la référence du bean. Cette valeur doit être identique à celle de l'attribut id du tag <jsp:useBean> utilisé pour instancier le bean.

L'attribut property= «*» permet d'alimenter automatiquement les propriétés du bean avec les paramètres correspondants contenus dans la requête. Le nom des propriétés et le nom des paramètres doivent être identiques.

Comme les paramètres de la requête sont toujours fournis sous forme de String, une conversion est réalisée en utilisant la méthode valueOf() du wrapper du type de la propriété.

Exemple :

```
<jsp:setProperty name="monBean" property="*" />
```

L'attribut property="*propertyNam*e" [param="*parameterNam*e"] permet de mettre à jour un attribut du bean. Par défaut, l'alimentation est faite automatiquement avec le paramètre correspondant dans la requête. Si le nom de la propriété et du paramètre sont différents, il faut préciser l'attribut property et l'attribut param qui doit contenir le nom du paramètre qui va alimenter la propriété du bean.

Exemple :

```
<jsp:setProperty name="monBean" property="nom"/>
```

L'attribut property="*propertyNam*e" value="{*string* | <%= *expression* %>}" permet d'alimenter la propriété du bean avec une valeur particulière.

Exemple :

```
<jsp:setProperty name="monBean" property="nom" value="toto" />
```

Il n'est pas possible d'utiliser param et value dans le même tag.

Exemple : Cette exemple est identique au précédent

```
<html>
<HEAD>
<TITLE>Essai d'instanciation d'un bean dans une JSP</TITLE>
</HEAD>
<body>
<p>Test d'utilisation d'un Bean dans une JSP </p>
<jsp:useBean id="personne" scope="request" class="test.Personne" />
<p>nom initial = <%= personne.getNom() %></p>
<jsp:setProperty name="personne" property="nom" value="mon nom" />
<p>nom mis à jour = <%= personne.getNom() %></p>
</body>
</html>
```

Ce tag peut être utilisé entre les tags <jsp:useBean> et </jsp:useBean> pour initialiser les propriétés du bean lors de son instanciation.

27.4.4.3. Le tag <jsp:getProperty>

Le tag <jsp:getProperty> permet d'obtenir la valeur d'un attribut d'un Bean. Le tag utilise le getter (méthode getXXX() ou XXX est le nom de la propriété) pour obtenir la valeur et l'insérer dans la page HTML généré. Le bean doit exister grâce à un appel au tag <jsp:useBean>.

La syntaxe est la suivante :

<jsp:getProperty name="*beanInstanceName*" property=" *propertyName*" />

L'attribut name indique le nom du bean tel qu'il a été déclaré dans le tag <jsp:useBean>.

L'attribut property indique le nom de la propriété dont on veut la valeur.

Exemple :

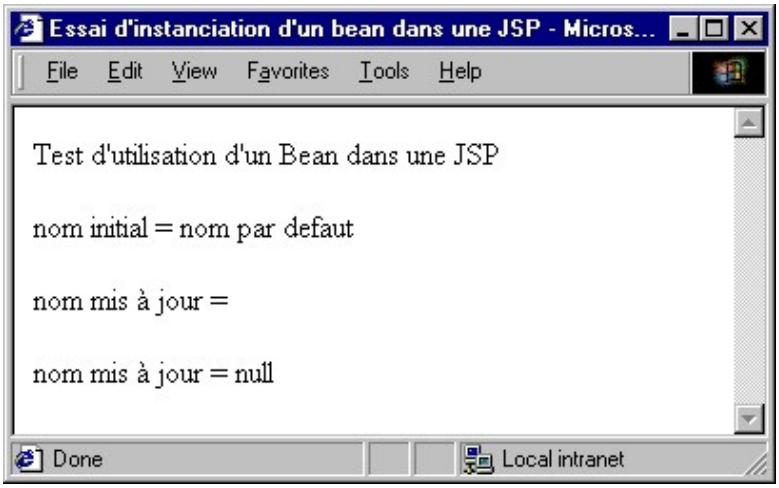
```
<html>
<HEAD>
<TITLE>Essai d'instanciation d'un bean dans une JSP</TITLE>
</HEAD>
<body>
<p>Test d'utilisation d'un Bean dans une JSP </p>
<jsp:useBean id="personne" scope="request" class="test.Personne" />
<p>nom initial = <jsp:getProperty name="personne" property="nom" /></p>
<jsp:setProperty name="personne" property="nom" value="mon nom" />
<p>nom mise à jour = <jsp:getProperty name="personne" property="nom" /></p>
</body>
</html>
```

Attention : ce tag ne permet pas d'obtenir la valeur d'une propriété indexée ni les valeurs d'un attribut d'un EJB.

Remarque : avec Tomcat 3.1, l'utilisation du tag <jsp:getProperty> sur un attribut dont la valeur est null n'affiche rien alors que l'utilisation d'un tag d'expression retourne « null ».

Exemple :

```
<html>
<HEAD>
<TITLE>Essai d'instanciation d'un bean dans une JSP</TITLE>
</HEAD>
<body>
<p>Test d'utilisation d'un Bean dans une JSP </p>
<jsp:useBean id="personne" scope="request" class="test.Personne" />
<p>nom initial = <jsp:getProperty name="personne" property="nom" /></p>
<% personne.setNom(null);%>
<p>nom mis à jour = <jsp:getProperty name="personne" property="nom" /></p>
<p>nom mis à jour = <%= personne.getNom() %></p>
</body>
</html>
```



27.4.4.4. Le tag de redirection <jsp:forward>

Le tag <jsp:forward> permet de rediriger la requête vers une autre URL pointant vers un fichier HTML, JSP ou un servlet.

Dès que le moteur de JSP rencontre ce tag, il redirige le requête vers l'URL précisée et ignore le reste de la JSP courante. Tout ce qui a été généré par la JSP est perdu.

La syntaxe est la suivante :

<jsp:forward page="{*relativeURL* | <%= *expression* %>}" />
ou
<jsp:forward page="{*relativeURL* | <%= *expression* %>}" >
<jsp:param name="*parameterName*" value="{ *parameterValue* | <%= *expression* %>}" /> +
</jsp:forward>

L'option page doit contenir la valeur de l'URL de la ressource vers laquelle la requête va être redirigée.

Cette URL est absolue si elle commence par un '/' ou relative à la JSP sinon. Dans le cas d'une URL absolue, c'est le serveur web qui détermine la localisation de la ressource.

Il est possible de passer un ou plusieurs paramètres vers la ressource appelée grâce au tag <jsp :param>.

Exemple :

```
Test8.jsp

<html>
<body>
<p>Page initiale appelée</p>
<jsp:forward page="forward.htm"/>
</body>
</html>
forward.htm
<HTML>
<HEAD>
<TITLE>Page HTML</TITLE>
</HEAD>
<BODY>
<p><table border="1" cellpadding="4" cellspacing="0" width="30%" align=center >
<tr bgcolor="#A6A5C2">
<td align="center">Page HTML forwardée</Td>
</Tr>
</table></p>
</BODY>
</HTML>
```

Le fichier forward.htm doit être dans le même répertoire que la JSP. Lors de l'appel à la JSP, c'est le page HTML qui est affichée. Le contenu généré par la page JSP n'est pas affiché.

27.4.4.5. Le tag <jsp:include>

Ce permet d'inclure le contenu d'un fichier dynamiquement au moment ou la servlet est executé. C'est la différence avec la directive include avec laquelle le fichier est inséré dans la JSP avant la génération de la servlet.

Cette section est en cours d'écriture

27.4.4.6. Le tag <jsp:plugin>

Cette section est en cours d'écriture

27.5. Un Exemple très simple

Exemple :

```
TestJSPIdent.html

<HTML>
<HEAD>
<TITLE>Identification</TITLE>
</HEAD>
<BODY>
<FORM METHOD=POST ACTION="jsp/TestJSPAccueil.jsp">
Entrer votre nom :
<INPUT TYPE=TEXT NAME="nom">
<INPUT TYPE=SUBMIT VALUE="SUBMIT">
</FORM>
</BODY>
</HTML>

TestJSPAccueil.jsp

<HTML>
<HEAD>
<TITLE>Accueil</TITLE>
</HEAD>
<BODY>
<%
String nom = request.getParameter("nom");
%>
<H2>Bonjour <%= nom %></H2>
</BODY>
</HTML>
```

27.6. L'utilisation d'une session

Cette section est en cours d'écriture

28. JSTL (Java server page Standard Tag Library)

28.1. Présentation

JSTL est l'acronyme de Java server page Standard Tag Library. C'est un ensemble de tags personnalisés développé sous la JSR 052 qui propose des fonctionnalités souvent rencontrées dans les JSP :

- Tag de structure (itération, conditionnement ...)
- Internationnalisation
- Exécution de requete SQL
- Utilisation de document XML

JSTL nécessite un conteneur d'application web qui implémente l'API servlet 2.3 et l'API JSP 1.2. L'implémentation de référence (JSTL−RI) de cette spécification est développée par le projet Taglibs du groupe Apache sous le nom « Standard ».

Il est possible de télécharger cette implémentation de référence à l'URL :
http://jakarta.apache.org/taglibs/doc/standard−doc/intro.html

JSTL est aussi inclus dans le JWSDP (Java Web Services Developer Pack), ce qui facilite son installation et son utilisation. Les exemples de cette section ont été réalisés avec le JWSDP 1.001

JSTL possède quatre bibliothèques de tag :

Rôle	TLD	Uri
Fonctions de base	c.tld	http://java.sun.com/jstl/core
Traitements XML	x.tld	http://java.sun.com/jstl/xml
Internationnalisation	fmt.tld	http://java.sun.com/jstl/fmt
Traitements SQL	sql.tld	http://java.sun.com/jstl/sql

JSTL propose un langage nommé EL (expression langage) qui permet de faire facilement référence à des objets java accessible dans les différents contexte de la JSP.

La bibliothèque de tag JSTL est livrée en deux versions :

- JSTL−RT : les expressions pour désigner des variables utilisant la syntaxe JSP classique
- JSTL−EL : les expressions pour désigner des variables utilisant le langage EL

Pour plus informations, il est possible de consulter les spécifications à l'url suivante :
http://jcp.org/aboutJava/communityprocess/final/jsr052/

28.2. Un exemple simple

Pour commencer, voici un exemple et sa mise en oeuvre détaillée. L'application web d'exemple se nomme test. Il faut créer un répertoire test dans le répertoire webapps de tomcat.

Pour utiliser JSTL, il faut copier les fichiers jstl.jar et standard.jar dans le répertoire WEB−INF/lib de l'application web.

Il faut copier les fichiers .tld dans le répertoire WEB−INF ou un de ces sous répertoires. Dans la suite de l'exemple, ces fichiers ont été placé le répertoire /WEB−INF/tld.

Il faut ensuite déclarer les bibliothèques à utiliser dans le fichier web.xml du répertoire WEB−INF comme pour toutes bibliothèque de tags personnalisés.

Exemple : pour la bibliothèque Core :

```
<taglib>
    <taglib−uri>http://java.sun.com/jstl/core</taglib−uri>
<taglib−location>/WEB−INF/tld/c.tld</taglib−location>
</taglib>
```

L'arborescence des fichiers est la suivante :

```
webapps
      test
            WEB−INF
                  lib
                        jstl.jar
                        standard.jar
                  tld
                        c.tld
                  web.xml
            test.jsp
```

Pour pouvoir utiliser une bibliothèque personnalisée, il faut utiliser la directive taglib :

Exemple :

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
```

Voici les code sources des différents fichiers de l'application web :

Exemple : fichier test.jsp

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
  <head>
    <title>Exemple</title>
  </head>

  <body>
      <c:out value="Bonjour" /><br/>
  </body>
</html>
```

Exemple : le fichier WEB−INF/web.xml

```
<?xml version="1.0" encoding="ISO−8859−1"?>

<!DOCTYPE web−app
    PUBLIC "−//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web−app23.dtd">
```

```
<web-app>
  <taglib>
    <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
<taglib-location>/WEB-INF/tld/c.tld</taglib-location>
</taglib>
</web-app>
```

Pour tester l'application, il suffit de lancer Tomcat et de saisir l'url localhost:8080/test/test.jsp dans un browser.

28.3. Le langage EL (Expression Langage)

JSTL propose un langage particulier constitué d'expressions qui permet d'utiliser et de faire référence à des objets java accessible dans les différents contexte de la page JSP. Le but est de fournir un moyen simple d'accéder aux données nécessaire à une JSP.

La syntaxe de base est ${xxx} ou xxx est le nom d'une variable d'un objet java définie dans un contexte particulier. La définition dans un contexte permet de définir la portée de la variable (page, requete, session ou application).

EL permet facilement de s'affranchir de la syntaxe de java pour obtenir une variable.

Exemple : accéder à l'attribut nom d'un objet personne situé dans la session avec Java

```
<%= session.getAttribute("personne").getNom() %>
```

Exemple : accéder à l'attribut nom d'un objet personne situé dans la session avec EL

```
${sessionScope.personne.nom}
```

EL possède par défaut les variables suivantes :

Variable	Rôle
PageScope	variable contenue dans la portée de la page (PageContext)
RequestScope	variable contenue dans la portée de la requète (HttpServletRequest)
SessionScope	variable contenue dans la portée de la session (HttpSession)
ApplicationScope	variable contenue dans la portée de l'application (ServletContext)
Param	paramètre de la requete http
ParamValues	paramètres de la requète sous la forme d'une collection
Header	en tête de la requète
HeaderValues	en têtes de la requète sous la forme d'une collection
InitParam	paramètre d'initialisation
Cookie	cookie
PageContext	objet PageContext de la page

EL propose aussi différents opérateurs :

Operateur	Rôle	Exemple
.	Obtenir une propriété d'un objet	${param.nom}
[]	Obtenir une propriété par son nom ou son indice	${param[« nom »]} ${row[1]}
Empty	Teste si un objet est null ou vide si c'est une chaine de caractère. Renvoie un booleen	${empty param.nom}
== eq	test l'égalité de deux objet	
!= ne	test l'inégalité de deux objet	
< lt	test strictement inférieur	
> gt	test strictement supérieur	
<= le	test inférieur ou égal	
>= ge	test supérieur ou égal	
+	Addition	
−	Soustraction	
*	Multiplication	
/ div	Division	
% mod	Modulo	
&& and		
|| or		
! not	Négation d'un valeur	

EL ne permet pas l'accès aux variables locales. Pour pouvoir accéder à de telles variables, il faut obligatoirement en créer une copie dans une des portées particulières : page, request, session ou application

Exemple :

```
<%
  int valeur = 101;
%>
  valeur = <c:out value="${valeur}" /><BR/>
```

Résultat :

```
valeur =
```

Exemple : avec la variable copiée dans le contexte de la page

```
<%
  int valeur = 101;
  pageContext.setAttribute("valeur", new Integer(valeur));
%>
  valeur = <c:out value="${valeur}" /><BR/>
```

Résultat :

```
valeur = 101
```

28.4. La bibliothèque Core

Elle propose les tags suivants répartis dans trois catégories :

Catégorie	Tag
Utilisation de EL	set out remove catch
Gestion du flux (condition et itération)	if choose forEach forTokens
Gestion des URL	import url redirect

Pour utiliser cette bibliothèque, il faut la déclarer dans le fichier web.xml du répertoire WEB−INF de l'application web.

Exemple :

```
 <taglib>
    <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
    <taglib-location>/WEB-INF/tld/c.tld</taglib-location>
</taglib>
```

Dans chaque JSP qui utilise un ou plusieurs tags de la bibliothèque, il faut la déclarer avec une directive taglib

Exemple :

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
```

28.4.1. Le tag set

Le tag set permet de stocker une variable dans une portée particulière (page, requete, session ou application).

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à stocker
target	nom de la variable contenant un bean dont la propriété doit être modifiée
property	nom de la propriété à modifier
var	nom de la variable qui va stocker la valeur
scope	portée de la variable qui va stocker la valeur

Exemple :

```
    <c:set var="maVariable1" value="valeur1" scope="page" />
    <c:set var="maVariable2" value="valeur2" scope="request" />
    <c:set var="maVariable3" value="valeur3" scope="session" />
    <c:set var="maVariable4" value="valeur4" scope="application" />
```

La valeur peut être déterminée dynamiquement.

Exemple :

```
    <c:set var="maVariable" value="${param.id}" scope="page" />
```

L'attribut target avec l'attribut property permet de modifier la valeur d'une propriété (précisée avec l'attribut property) d'un objet (précisé avec l'attribut target).

La valeur de la variable peut être précisée dans le corps du tag plutot que d'utiliser l'attribut value.

Exemple :

```
<c:set var="maVariable" scope="page">
       Valeur de ma variable
</c:set>
```

28.4.2. Le tag out

Le tag out permet d'envoyer dans le flux de sortie de la JSP le résultat de l'évaluation de l'expression fournie dans le paramètre « value ». Ce tag est équivalent au tag d'expression <%= … %> de JSP.

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à afficher (obligatoire)
default	définir une valeur par défaut si la valeur est null

escapeXml	booléen qui precise si les caractères particuliers (< > & ...) doivent être convertis en leur équivalent HTML (< > &amp ; …)

Exemple :

```
    <c:out value='${pageScope.maVariable1}' />
    <c:out value='${requestScope.maVariable2}' />
    <c:out value='${sessionScope.maVariable 3}' />
    <c:out value='${applicationScope.maVariable 4}' />
```

Il n'est pas obligatoire de préciser la portée dans laquelle la variable est stockée : dans ce cas, la variable est recherchée prioritairement dans la page, la requete, la session et enfin l'application.

L'attribut default permet de définir une valeur par défaut si le résultat de l'évaluation de la valeur est null. Si la valeur est null et que l'attribut default n'est pas utilisé alors c'est une chaine vide qui est envoyée dans le flux de sortie.

Exemple :

```
   <c:out value="${personne.nom}" default="Inconnu" />
```

Le tag out est particulière utile pour générer le code dans un formulaire en remplaçant avantageusement les scriplets.

Exemple :

```
<input type="text" name="nom" value="<c:out value="${param.nom}"/>" />
```

28.4.3. Le tag remove

Le tag remove permet de supprimer une variable d'une portée particulière.

Il possède plusieurs attributs :

Attribut	Rôle
var	nom de la variable à supprimer (obligatoire)
scope	portée de la variable

Exemple :

```
    <c:remove var="maVariable1" scope="page" />
    <c:remove var="maVariable2" scope="request" />
    <c:remove var="maVariable3" scope="session" />
    <c:remove var="maVariable4" scope="application" />
```

28.4.4. Le tag catch

Ce tag permet de capturer des exceptions qui sont levées lors de l'éxécution du code inclus dans son corps.

Il possède un attribut :

Attribut	Rôle
var	nom d'une variable qui va contenir des informations sur l'anomalie

Si l'attribut var n'est pas utilisé, alors toutes les exceptions levées lors de l'exécution du corps du tag sont ignorées.

Exemple : code non protégé

```
<c:set var="valeur" value="abc" />
<fmt:parseNumber var="valeurInt" value="${valeur}"/>
```

Résultat : une exception est lévée

```
javax.servlet.ServletException: In <parseNumber>, value attribute can not be parsed: "abc"
        at org.apache.jasper.runtime.PageContextImpl.handlePageException(PageContextImpl.java:471)
        at org.apache.jsp.test$jsp.jspService(test$jsp.java:1187)
        at org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:107)
```

L'utilisation du tag catch peut empecher le plantage de l'application.

Exemple :

```
<c:set var="valeur" value="abc" />
<c:catch var="erreur">
   <fmt:parseNumber var="valeurInt" value="${valeur}"/>
</c:catch>
<c:if test="${not empty erreur}">
   la valeur n'est pas numerique
</c:if>
```

Résultat :

```
la valeur n'est pas numerique
```

L'objet désigné par l'attribut var du tag catch possède une propriété message qui contient le message d'erreur

Exemple :

```
<c:set var="valeur" value="abc" />
<c:catch var="erreur">
   <fmt:parseNumber var="valeurInt" value="${valeur}"/>
</c:catch>
<c:if test="${not empty erreur}">
 <c:out value="${erreur.message}"/>
</c:if>
```

Résultat :

```
In &lt;parseNumber&gt;, value attribute can not be parsed: "abc"
```

Le soucis avec ce tag est qu'il n'est pas possible de savoir qu'elle exception a été levée.

28.4.5. Le tag if

Ce tag permet d'évaluer le contenu de son corps si la condition qui lui est fournie est vrai.

Il possède plusieurs attributs :

Attribut	Rôle
test	condition à évaluer
var	nom de la variable qui contiendra le résultat de l'évaluation
scope	portée de la variabe qui contiendra le résultat

Exemple :

```
<c:if test="${empty personne.nom}" >Inconnu</c:if>
```

Le tag peut ne pas avoir de corps si le tag est simplement utiliser pour stocker le résultat de l'évaluation de la condition dans une variable.

Exemple :

```
<c:if test="${empty personne.nom}" var="resultat" />
```

Le tag if est particulière utile pour générer le code dans un formulaire en remplaçant avantageusement les scriplets.

Exemple : selection de la bonne occurrence dont la valeur est fournie en paramètre de la requète

```
<FORM NAME="form1" METHOD="post" ACTION="">
  <SELECT NAME="select">
    <OPTION VALUE="choix1" <c:if test="${param.select == 'choix1'}" >selected</c:if> >
      choix 1</OPTION>
    <OPTION VALUE="choix2" <c:if test="${param.select == 'choix2'}" >selected</c:if> >
      choix 2</OPTION>
    <OPTION VALUE="choix3" <c:if test="${param.select == 'choix3'}" >selected</c:if> >
      choix 3</OPTION>
  </SELECT>
</FORM>
```

Pour tester le code, il faut fournir en paramètre dans l'url select=choix2

Exemple :

```
http://localhost:8080/test/test.jsp?select=choix2
```

28.4.6. Le tag choose

Ce tag permet de traiter différents cas mutuellements exclusifs dans un même tag. Le tag choose ne possède pas d'attribut. Il doit cependant posséder un ou plusieurs tags fils « when ».

Le tag when possède l'attribut test qui permet de préciser la condition à évaluer. Si la condition est vrai alors le corps du tag when est évaluer et le résultat est envoyé dans le flux de sortie de la JSP

Le tag otherwise permet de définir un cas qui ne correspond à aucun des autres inclus dans le tag. Ce tag ne possède aucun attribut.

Exemple :

```
<c:choose>
  <c:when test="${personne.civilite == 'Mr'}">
    Bonjour Monsieur
  </c:when>
  <c:when test="${personne.civilite == 'Mme'}">
    Bonjour Madame
  </c:when>
  <c:when test="${personne.civilite == 'Mlle'}">
    Bonjour Mademoiselle
  </c:when>
  <c:otherwise>
    Bonjour
  </c:otherwise>
</c:choose>
```

28.4.7. Le tag forEach

Ce tag permet de parcourir les différents éléments d'une collection et ainsi d'exécuter de façon répétitive le contenu de son corps.

Il possède plusieurs attributs :

Attribut	Rôle
var	nom de la variable qui contient l'élément en cours de traitement
items	collection à traiter
varStatus	nom d'un variable qui va contenir des informations sur l'itération en cours de traitement
begin	numéro du premier élément à traiter (le premier possède le numéro 0)
end	numéro du dernier élément à traiter
step	pas des éléments à traiter (par défaut 1)

A chaque itération, la valeur de la variable dont le nom est précisé par la propriété var change pour contenir l'élement de la collection en cours de traitement.

Aucun des attributs n'est obligatoire mais il faut obligatoirement qu'il y ait l'attribut items ou les attributs begin et end.

Le tag forEach peut aussi réaliser des itérations sur les nombres et non sur des éléments d'une collection. Dans ce cas, il ne faut pas utiliser l'attributs items mais uniquement utiliser les attributs begin et end pour fournir les bornes inférieures et supérieures de l'itération.

Exemple :

```
<c:forEach begin="1" end="4" var="i">
<c:out value="${i}"/><br>
</c:forEach>
```

Résultat :

```
1
2
3
4
```

L'attribut step permet de préciser le pas de l'itération.

Exemple :

```
<c:forEach begin="1" end="12" var="i" step="3">
  <c:out value="${i}"/><br>
</c:forEach>
```

Exemple :

```
1
4
7
10
```

L'attribut varStatus permet de définir une variable qui va contenir des informations sur l'itération en cours d'éxécution. Cette variable possède plusieurs propriétés :

Attribut	Rôle
index	indique le numéro de l'occurrence dans l'ensemble de la collection
count	indique le numéro de l'itération en cours (en commençant par 1)
first	booléen qui indique si c'est la première itération
last	booléen qui indique si c'est la dernière itération

Exemple :

```
<c:forEach begin="1" end="12" var="i" step="3" varStatus="vs">
  index = <c:out value="${vs.index}"/> :
  count = <c:out value="${vs.count}"/> :
  value = <c:out value="${i}"/>
  <c:if test="${vs.first}">
     : Premier element
  </c:if>
  <c:if test="${vs.last}">
     : Dernier element
  </c:if>
  <br>
</c:forEach>
```

Résultat :

```
  index = 1 : count = 1 : value = 1 : Premier element
  index = 4 : count = 2 : value = 4
  index = 7 : count = 3 : value = 7
  index = 10 : count = 4 : value = 10 : Dernier element
```

28.4.8. Le tag forTokens

Ce tag permet de découper une chaine selon un ou plusieurs séparateurs donnés et ainsi d'exécuter de façon répétitive le contenu de son corps autant de fois que d'occurrences trouvées.

Il possède plusieurs attributs :

Attribut	Rôle
var	variable qui contient l'occurrence en cours de traitement (obligatoire)
items	la chaine de caractères à traiter (obligatoire)
delims	précise le séparateur
varStatus	nom d'un variable qui va contenir des informations sur l'itération en cours de traitement
begin	numero du premier élément à traiter (le premier possède le numéro 0)
end	numéro du dernier élément à traiter
step	pas des éléments à traiter (par défaut 1)

L'attribut delims peut avoir comme valeur une chaine de caractères ne contenant qu'un seul caractère (délimiteur unique) ou un ensemble de caractères (délimiteurs multiples).

Exemple :

```
<c:forTokens var="token" items="chaine 1;chaine 2;chaine 3" delims=";">
  <c:out value="${token}" /><br>
</c:forTokens>
```

Exemple :

```
chaine 1
chaine 2
chaine 3
```

Dans le cas ou il y a plusieurs délimiteurs, chacun peut servir de séparateur

Exemple :

```
<c:forTokens var="token" items="chaine 1;chaine 2,chaine 3" delims=";,">
  <c:out value="${token}" /><br>
</c:forTokens>
```

Attention : Il n'y a pas d'occurrence vide. Dans le cas ou deux séparateurs se suivent consécutivement dans la chaine à traiter, ceux ci sont considérés comme un seul séparateur. Si la chaine commence ou se termine par un séparateur, ceux ci sont ignorés.

Exemple :

```
<c:forTokens var="token" items="chaine 1;;chaine 2;;;chaine 3" delims=";">
  <c:out value="${token}" /><br>
</c:forTokens>
```

Résultat :

```
chaine 1
chaine 2
chaine 3
```

Il est possible de ne traiter qu'un sous ensemble des occurrences de la collection. JSTL attribut à chaque occurrence un numéro incrémenter de 1 en 1 à partir de 0. Les attributs begin et end permettent de préciser une plage d'occurrence à traiter.

Exemple :

```
<c:forTokens var="token" items="chaine 1;chaine 2;chaine 3" delims=";" begin="1" end="1" >
  <c:out value="${token}" /><br>
</c:forTokens>
```

Résultat :

```
chaine 2
```

Il est possible de n'utiliser que l'attribut begin ou l'attribut end. Si seul l'attribut begin est précisé alors les n dernières occurrences seront traitées. Si seul l'attribut end est précisé alors seuls les n premières occurrences seront traitées.

Les attributs varStatus et step ont le même rôle que ceux du tag forEach.

28.4.9. Le tag import

Ce tag permet d'accéder à une ressource via son URL pour l'inclure ou l'utiliser dans les traitements de la JSP. La ressource accédée peut être dans une autre application.

Son grand intérêt par rapport au tag <jsp :include> est de ne pas être limité au contexte de l'application web.

Il possède plusieurs attributs :

Attribut	Rôle
url	url de la ressource (obligatoire)
var	nom de la variable qui va stocker le contenu de la ressource sous la forme d'une chaine de caractère
scope	portée de la variable qui va stocker le contenu de la ressource
context	contexte de l'application web qui contient la ressource (si la ressource n'est pas l'application web courante)
charEncoding	jeux de caractères utilisé par la ressource
varReader	nom de la variable qui va stocker le contenu de la ressource sous la forme d'un objet de type java.io.Reader

L'attribut url permet de préciser l'url de la ressource. Cette url peut être relative (par rapport à l'application web) ou absolue.

Exemple :

```
<c:import url="/message.txt" /><br>
```

Par défaut, le contenu de la ressource est inclus dans la JSP. Il est possible de stocker le contenu de la ressource dans une chaine de caractères en utilisant l'attribut var. Cet attribut attend comme valeur le nom de la variable.

Exemple :

```
<c:import url="/message.txt" var="message" />
<c:out value="${message}" /><BR/>
```

28.4.10. Le tag redirect

Ce tag permet de faire une redirection vers une nouvelle URL.

Les paramètres peuvent être fournis grace à un ou plusieurs tags fils param.

Exemple :

```
<c:redirect url="liste.jsp">
  <c:param name="id" value="123"/>
</c:redirect>
```

28.4.11. Le tag url

Ce tag permet de formatter une url. Il possède plusieurs attributs :

Attribut	Rôle
value	base de l'url (obligatoire)
var	nom de la variable qui va stocker l'url
scope	portée de la variable qui va stocker l'url
context	

Le tag url peut avoir un ou plusieurs tag fils « param ». Le tag param permet de préciser un parametre et sa valeur pour qu'il soit ajouté à l'url générée.

Le tag param possède deux attributs :

Attribut	Rôle
name	nom du paramètre
value	valeur du paramètre

Exemple :

```
<a href="<c:url url="/index.jsp"/>"/>
```

28.5. La bibliothèque XML

Cette bibliothèque permet de manipuler des données en provenance d'un document XML.

Elle propose les tags suivants répartis dans trois catégories :

Catégorie	Tag
Fondamentale	parse set out
Gestion du flux (condition et itération)	if choose forEach
Transformation XSLT	transform

Les exemples de cette section utilisent un fichier xml nommé personnes.xml dont le contenu est le suivant :

Fichier utilisé dans les exemples :

```
<personnes>
  <personne id="1">
    <nom>nom1</nom>
    <prenom>prenom1</prenom>
  </personne>
  <personne id="2">
    <nom>nom2</nom>
    <prenom>prenom2</prenom>
  </personne>
  <personne id="3">
    <nom>nom3</nom>
    <prenom>prenom3</prenom>
  </personne>
</personnes>
```

L'attribut select des tags de cette bibliothèque utilise la norme Xpath pour sa valeur. JSTL propose une extension supplémentaire à Xpath pour préciser l'objet sur lequel l'expression doit être évaluée. Il suffit de préfixer le nom de la variable par un $

Exemple : recherche de la personne dont l'id est 2 dans un objet nommé listepersonnes qui contient l'arborescence du document xml.

```
$listepersonnes/personnes/personne[@id=2]
```

L'implémentation de JSTL fournie avec le JWSDP utilise Jaxen comme moteur d'interprétation XPath. Donc pour utiliser cette bibliothèque, il faut s'assurer que les fichiers saxpath.jar et jaxen−full.jar soient présents dans le répertoire lib du répertoire WEB−INF de l'application web.

Pour utiliser cette bibliothèque, il faut la déclarer dans le fichier web.xml du répertoire WEB−INF de l'application web.

Exemple :

```
  <taglib>
    <taglib−uri>http://java.sun.com/jstl/xml</taglib−uri>
    <taglib−location>/WEB−INF/tld/x.tld</taglib−location>
  </taglib>
```

Dans chaque JSP qui utilise un ou plusieurs tags de la bibliothèque, il faut la déclarer avec une directive taglib.

Exemple :

```
<%@ taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>
```

28.5.1. Le tag parse

La tag parse permet d'analyser un document et de stocker le résultat dans une variable qui pourra être exploité par la JSP ou une autre JSP selon la portée sélectionnée pour le stockage.

Attribut	Rôle
xml	contenu du document à analyser
var	nom de la variable qui va contenir l'arbre DOM générer par l'analyse
scope	portée de la variable qui va contenir l'arbre DOM
varDom	
scopeDom	
filter	
System	

Exemple : chargement et sauvegarde dans une variable de la JSP

```
<c:import url="/personnes.xml" var="personnes" />
<x:parse xml="${personnes}" var="listepersonnes"  />
```

Dans cette exemple, il suffit simplement que le fichier personnes.xml soit dans le dossier racine de l'application web.

28.5.2. Le tag set

Le tag set est équivalent au tag set de la bibliothèque core. Il permet d'évaluer l'expression Xpath fournie dans l'attribut select et de placer le résultat de cette évaluation dans une variable. L'attribut var permet de préciser la variable qui va recevoir le résultat de l'évaluation sous la forme d'un noeud de l'arbre du document XML.

Il possède plusieurs attributs :

Attribut	Rôle
select	expression XPath à évaluer
var	nom de la variable qui va stocker le résultat de l'évaluation
scope	portée de la variable qui va stocker le résultat

Exemple : obtenir les informations de la personne dont l'id est 2

```
<c:import url="/personnes.xml" var="personnes" />
<x:parse xml="${personnes}" var="listepersonnes"  />
```

```
<x:set var="unepersonne" select="$listepersonnes/personnes/personne[@id=2]" />
<h1>nom = <x:out select="$unepersonne/nom"/></h1>
```

28.5.3. Le tag out

Le tag out est équivalent au tag out de la bibliothèque core. Il est permet d'évaluer l'expression Xpath fournie dans l'attribut select et d'envoyer le résultat dans le flux de sortie. L'attribut select permet de préciser l'expression Xpath qui doit être évaluée.

Il possède plusieurs attributs :

Attribut	Rôle
select	expression XPath à évaluer
escapeXML	

Exemple : Afficher le nom de la personne dont l'id est 2

```
<c:import url="/personnes.xml" var="personnes" />
<x:parse xml="${personnes}" var="listepersonnes"  />
<x:set var="unepersonne" select="$listepersonnes/personnes/personne[@id=2]" />
<h1><x:out select="$unepersonne/nom"/></h1>
```

Pour stocker le résultat de l'évaluation d'une expression dans une variable, il faut utiliser une combinaison du tag x:out et c:set

Exemple :

```
<c:set var="personneId">
      <x:out select="$listepersonnes/personnes/personne[@id=2]" />
    </c:set>
```

28.5.4. Le tag if

Ce tag est équivalent au tag if de la bibliothèque core sauf qu'il évalue une expression XPath

Il possède plusieurs attributs :

Attribut	Rôle
select	expression XPath à évaluer sous la forme d'un booléen
var	nom de la variable qui va stocker le résultat de l'évaluation
scope	portée de la variable qui va stocker le résultat de l'évaluation

28.5.5. La tag choose

Ce tag est équivalent au tag choose de la bibliothèque core sauf qu'il évalue des expressions XPath

28.5.6. Le tag forEach

Ce tag est équivalent au tag forEach de la bibliothèque Core. Il permet de parcourir les noeuds issus de l'évaluation d'une expression Xpath.

Il possède plusieurs attributs :

Attribut	Rôle
select	expression XPath à évaluer (obligatoire)
var	nom de la variable qui va contenir le noeud en cours de traitement

Exemple : parcours des personnes et affichage de l'id, du nom et du prénom

```
<c:import url="/personnes.xml" var="personnes" />
<x:parse xml="${personnes}" var="listepersonnes"  />
<x:forEach var="unepersonne" select="$listepersonnes/personnes/*">
    <c:set var="personneId">
      <x:out select="$unepersonne/@id"/>
    </c:set>
    <c:out value="${personneId}" /> – <x:out select="$unepersonne/nom"/>  
       <x:out select="$unepersonne/prenom"/> <br>
</x:forEach>
```

28.5.7. Le tag transform

Ce tag permet d'appliquer une transformation XSLT à un document XML. L'attribut xsl permet de préciser la feuille de style XSL. L'attribut optionnel xml permet de préciser le document xml.

Il possède plusieurs attributs :

Attribut	Rôle
xslt	feuille se style XSLT (obligatoire)
xml	nom de la variable qui contient le document XML à traiter
var	nom de la variable qui va recevoir le résultat de la transformation
scope	portée de la variable qui va recevoir le résultat de la transformation
xmlSystemId	
xsltSystemId	
result	

Exemple :

```
<x:transform xml='${docXml}' xslt='${feuilleXslt}'/>
```

Le document xml à traiter peut être fourni dans le corps du tag

Exemple :

```
<x:transform xslt='${feuilleXslt}'>
  <personnes>
    <personne id="1">
      <nom>nom1</nom>
      <prenom>prenom1</prenom>
    </personne>
    <personne id="2">
      <nom>nom2</nom>
      <prenom>prenom2</prenom>
    </personne>
    <personne id="3">
      <nom>nom3</nom>
      <prenom>prenom3</prenom>
    </personne>
  </personnes>
</x:transform>
```

Le tag transform peut avoir un ou plusieurs noeuds fils param pour fournir des paramètres à la feuille de style XSLT.

28.6. La bibliothèque I18n

Cette bibliothèque facilite l'internationnalisation d'une page JSP.

Elle propose les tags suivants répartis dans trois catégories :

Catégorie	Tag
Définition de la langue	setLocale
Formattage de messages	>bundle message setBundle
Formattage de dates et nombres	formatNumber parseNumber formatDate parseDate setTimeZone timeZone

Pour utiliser cette bibliothèque, il faut la déclarer dans le fichier web.xml du répertoire WEB−INF de l'applciation web.

Exemple :

```
  <taglib>
    <taglib-uri>http://java.sun.com/jstl/fmt</taglib-uri>
<taglib-location>/WEB-INF/tld/fmt.tld</taglib-location>
</taglib>
```

Dans chaque JSP qui utilise un ou plusieurs tags de la bibliothèque, il faut la déclarer avec une directive taglib

Exemple :

```
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
```

Le plus simple pour mettre en oeuvre la localisation des messages, c'est de définir un ensemble de fichier qui sont appelé bundle en anglais.

Il faut définir un fichier pour la langue par défaut et un fichier pour chaque langue particulière. Tous ces fichiers ont un préfix commun appelé basename et doivent avoir comme extension .properties. Les fichiers pour les langues particulieres doivent le préfix commun suivit d'un underscore puis du code langue et eventuellement d'un underscore suivi du code pays. Ces fichiers doivent être inclus dans le classpath : le plus simple est de les copier dans le répertoire WEB−INF/classes de l'application web.

Exemple :

```
message.properties
messageen.properties
```

Dans chaque fichier, les clés sont identiques, seul la valeur associé à la clé change.

Exemple : le fichier message.properties pour le français (langue par défaut)

```
msg=bonjour
```

Exemple : le fichier messageen.properties pour l'anglais

```
msg=Hello
```

Pour plus d'information, voir le chapitre sur l'internationalisation.

28.6.1. Le tag bundle

Ce tag permet de préciser un bundle à utiliser dans les traitements contenus dans son corps.

Il possède plusieurs attributs :

Attribut	Rôle
baseName	nom de base de ressource à utiliser (obligatoire)
prefix	

Exemple :

```
<fmt:bundle basename="message" >
  <fmt:message key="msg"/>
</fmt:bundle>
```

28.6.2. Le tag setBundle

Ce tag permet de forcer le bundle à utiliser par défaut.

Il possède plusieurs attributs :

Attribut	Rôle
baseName	nom de base de ressource à utiliser (obligatoire)
var	nom de la variable qui va stocker le nouveau bundle
scope	portée de la variable qui va recevoir le nouveau bundle

Exemple :

```
mon message =
<fmt:setBundle basename="message" />
  <fmt:message key="msg"/>
```

28.6.3. Le tag message

Ce tag permet de localiser un message.

Il possède plusieurs attributs :

Attribut	Rôle
key	clé du message à utiliser
bundle	bundle à utiliser
var	nom de la variable qui va recevoir le résultat du formattage
scope	portée de la variable qui va recevoir le résutlat du formattage

Pour fournir chaque valeur, il faut utiliser un ou plusieurs tags fils param pour fournir la valeur correspondante.

Exemple :

```
mon message =
<fmt:setBundle basename="message" />
  <fmt:message key="msg"/>
```

Résultat :

```
mon message = bonjour
```

Si aucune valeur n’est trouvée pour la clé fournie alors le tag renvoie ???XXX ??? ou XXX représente le nom de la clé.

Exemple :

```
mon message =
<fmt:setBundle basename="message" />
  <fmt:message key="test"/>
```

Résultat :

```
mon message = ???test???
```

28.6.4. Le tag setLocale

Ce tag permet de sélectionner une nouvelle Locale.

Exemple :

```
<fmt:setLocale value="en"/>
mon message =
<fmt:setBundle basename="message" />
  <fmt:message key="msg"/>
```

Résultat :

```
mon message = Hello
```

28.6.5. Le tag formatNumber

Ce tag permet de formatter des nombres selon la locale. L'attribut value permet de préciser la valeur à formatter. L'attribut type permet de préciser le type de formattage à réaliser.

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à formatter
type	CURRENCY ou NUMBER ou PERCENT
pattern	format personnalisé
currencyCode	code de la monnaie à utiliser pour le type CURRENCY
currencySymbol	symbole de la monnaire à utiliser pour le type CURRENCY
groupingUsed	booléen pour préciser si les nombre doivent être groupés
maxIntegerDigits	nombre maximun de chiffre dans la partie entière
minIntegerDigits	nombre minimum de chiffre dans la partie entière
maxFractionDigits	nombre maximun de chiffre dans la partie décimale
minFractionDigits	nombre minimum de chiffre dans la partie décimale
var	nom de la variable qui va stocker le résultat
scope	portée de la variable qui va stocker le résultat

Exemple :

```
<c:set var="montant" value="12345.67" />
montant = <fmt:formatNumber value="${montant}" type="currency"/>
```

28.6.6. Le tag parseNumber

Ce tag permet de convertir une chaine de caractère qui contient un nombre en une variable décimale.

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à traiter
type	CURRENCY ou NUMBER ou PERCENT
parseLocale	Locale à utiliser lors du traitement
integerOnly	booléen qui indique si le résultat doit être un entier (true) ou un flottant (false)
pattern	format personnalisé
var	nom de la variable qui va stocker le résultat
scope	portée de la variable qui va stocker le résultat

Exemple : convertir en entier un identifiant passé en paramètre de la requète

```
<fmt:parseNumber value="${param.id}" var="id"/>
```

28.6.7. Le tag formatDate

Ce tag permet de formatter des dates selon la locale.

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à formatter
type	DATE ou TIME ou BOTH
dateStyle	FULL ou LONG ou MEDIUM ou SHORT ou DEFAULT
timeStyle	FULL ou LONG ou MEDIUM ou SHORT ou DEFAULT
pattern	format personnalisé
timeZone	timeZone utilisé pour le formattage
var	nom de la variable qui va stocker le résultat
scope	portée de la variable qui va stocker le résultat

L'attribut value permet de préciser la valeur à formatter. L'attribut type permet de préciser le type de formattage à réaliser. L'attribut dateStyle permet de préciser le style du formattage.

Exemple :

```
<jsp:useBean id="now" class="java.util.Date" />
Nous sommes le <fmt:formatDate value="${now}" type="date" dateStyle="full"/>.
```

28.6.8. Le tag parseDate

Ce tag permet d'analyser une chaine de caractères contenant une date pour créer un objet de type java.util.Date.

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à traiter
type	DATE ou TIME ou BOTH
dateStyle	FULL ou LONG ou MEDIUM ou SHORT ou DEFAULT
timeStyle	FULL ou LONG ou MEDIUM ou SHORT ou DEFAULT
pattern	format personnalisé
parseLocale	Locale utilisé pour le formattage
timeZone	timeZone utilisé pour le formattage
var	nom de la variable de type java.util.date qui va stocker le résultat
scope	portée de la variable qui va stocker le résultat

28.6.9. Le tag setTimeZone

Ce tag permet de stocker un fuseau horraire dans une variable.

Il possède plusieurs attributs :

Attribut	Rôle
value	fuseau horraire à stocker (obligatoire)
var	nom de la variable de stockage
scope	portée de la variable de stockage

28.6.10. Le tag timeZone

Ce tag permet de préciser un fuseau horraire particulier à utiliser dans son corps.

Il possède plusieurs attributs :

Attribut	Rôle
value	chaine de caractère ou objet java.util.TimeZone qui précise le fuseau horraire à utiliser

28.7. La bibliothèque Database

Cette biliothèque facilite l'accès aux bases de données. Son but n'est pas de remplacer les accès réalisés grace à des beans ou des EJB mais de fournir une solution simple mais non robuste pour accéder à des bases de données. Ceci est cependant particulièrement utile pour développer des pages de tests ou des prototypes.

Elle propose les tags suivants répartis dans deux catégories :

Catégorie	Tag
Définition de la source de données	setDataSource
Execution de requete SQL	query transaction update

Pour utiliser cette bibliothèque, il faut la déclarer dans le fichier web.xml du répertoire WEB−INF de l'application web.

Exemple :

```
  <taglib>
    <taglib−uri>http://java.sun.com/jstl/sql</taglib−uri>
    <taglib−location>/WEB−INF/tld/sql.tld</taglib−location>
</taglib>
```

Dans chaque JSP qui utilise un ou plusieurs tags de la bibliothèque, il faut la déclarer avec une directive taglib

Exemple :

```
<%@ taglib uri="http://java.sun.com/jstl/sql" prefix="sql" %>
```

28.7.1. Le tag setDataSource

Ce tag permet de créer une connection vers la base de données à partir des données fournies dans les différents attributs du tag.

Il possède plusieurs attributs :

Attribut	Rôle
driver	nom de la classe du pilote JDBC à utiliser
source	url de la base de données à utiliser
user	nom de l'utilisateur à utiliser lors de la connection
password	mot de passe de l'utilisateur à utiliser lors de la connection
var	nom de la variable qui va stocker l'objet créer lors de la connection
scope	portée de la variable qui va stocker l'objet créer
dataSource	

Exemple : accéder à une base via ODBC dont le DNS est test
`<sql:setDataSource driver="sun.jdbc.odbc.JdbcOdbcDriver" url="jdbc:odbc:test" user="" password=""/>`

28.7.2. Le tag query

Ce tag permet de réaliser des requètes de sélection sur une source de données.

Il possède plusieurs attributs :

Attribut	Rôle
sql	requète SQL à executer
var	nom de la variable qui stocke les résultats de l'exécution dela requète
scope	portée de la variable qui stocke le résultat
startRow	numéro de l'occurrence de départ à traiter
maxRow	nombre maximum d'occurrence à stocker
dataSource	connection particulière à la base de données à utiliser

L'attribut sql permet de préciser la requète à executer :

Exemple :
`<sql:query var="reqPersonnes" sql="SELECT * FROM personnes" />`

Le résultat de l'éxécution de la requète est stocké dans un objet qui implémente l'interface `javax.servlet.jsp.jstl.sql.Result dont le nom est donné via l'attribut var`

L'interface Result possède cinq getter :

Méthode	Rôle
String[] getColumnNames()	renvoie un tableau de chaines de caractères qui contient le nom des colonnes
int getRowCount()	renvoie le nombre d'enregistrements trouvé lors de l'éxécution de la requète
Map[] getRows()	renvoie une collection qui associe à chaque colonne la valeur associé pour l'occurrence en cours
Object[][] getRowsByIndex()	renvoie un tableau contenant les colonnes et leur valeur
boolean isLimitedByMaxRows()	renvoie un booléen qui indique si le résultat de la requète a été limité

Exemple : connaitre le nombre d'occurrence renvoyées par la requète
`<p>Nombre d'enregistrement trouvé : <c:out value="${reqPersonnes.rowCount}" /></p>`

La requète SQL peut être précisée avec l'attribut sql ou dans le corps du tag

Exemple :

```
<sql:query var="reqPersonnes" >
  SELECT * FROM personnes
</sql:query>
```

Le tag forEach de la bibliothèque core est particulièrement utile pour itérer sur chaque occurrence retournée par la requete SQL.

Exemple :

```
<TABLE border="1" CELLPadding="4" cellspacing="0">
<TR>
<td>id</td>
<td>nom</td>
<td>prenom</td>
</TR>

<c:forEach var="row" items="${reqPersonnes.rows}" >
<TR>
<td><c:out value="${row.id}" /></td>
<td><c:out value="${row.nom}" /></td>
<td><c:out value="${row.prenom}" /></td>
</TR>
</c:forEach>
</TABLE>
```

Il est possible de fournir des valeurs à la requète SQL. Il faut remplacer dans la requète SQL la valeur par le caractère ?. Pour fournir, la ou les valeurs il faut utiliser un ou plusieurs tags fils param.

Le tag param possède un seul attribut :

Attribut	Rôle
value	valeur de l'occurrence correspondante dans la requete SQL

Pour les valeurs de type date, il faut utiliser le tag dateParam.

Le tag dateParam possède plusieurs attributs :

Attribut	Rôle
value	objet de type java.util.date qui contient la valeur de la date (obligatoire)
type	format de la date : TIMESTAMP ou DATE ou TIME

Exemple :

```
<c:set var="id" value="2" />

<sql:query var="reqPersonnes" >
  SELECT * FROM personnes where id = ?
            <sql:param value="${id}" />
</sql:query>
```

28.7.3. Le tag transaction

Ce tag permet d'encapsuler plusieurs requètes SQL dans une transaction.

Il possède plusieurs attributs :

Attribut	Rôle
dataSource	connection particulière à la base de données à utiliser
isolation	READCOMMITTED ou READUNCOMMITTED ou REPEATABLEREAD ou SERIALIZABLE

28.7.4. Le tag update

Ce tag permet de réaliser une mise à jour grace à une requete SQL sur la source de données.

Il possède plusieurs attributs :

Attribut	Rôle
sql	requète SQL à exécuter
var	nom de la variable qui stocke le nombre d'occurrence impactée par l'exécution dela requète
scope	portée de la variable qui stocke le nombre d'occurrence impactée
dataSource	connection particulière à la base de données à utiliser

Exemple :

```
<c:set var="id" value="2" />
<c:set var="nouveauNom" value="nom 2 modifié" />

<sql:update var="nbRec">
UPDATE personnes
SET nom = ?
WHERE id=?
lt;sql:param value="${nouveauNom}"/>
<sql:param value="${id}"/>
</sql:update>

<p>nb enregistrement modifiés = <c:out value="${nbRec}"/></p>
```

29. Les frameworks

Ce chapitre est en cours d'écriture

30. Java et XML

L'utilisation ensemble de Java et XML est facilité par le fait qu'ils ont plusieurs points communs :

- indépendance de toute plateforme
- conçu pour être utilisé sur un réseau
- prise en charge de la norme Unicode

30.1. Présentation de XML

XML est l'acronyme de « eXtensible Markup Language ».

XML permet d'échanger des données entres applications hétérogènes car il permet de modéliser et de stocker des données de façon portable.

XML est extensible dans la mesure ou il n'utilise pas de tags prédéfinis comme HTML et il permet de définir de nouvelles balises : c'est un métalangage.

Le format HTML est utilisé pour formater et afficher les données qu'il contient : il est destiné à structurer, formater et échanger des documents d'une façon la plus standard possible..

XML est utilisé pour modéliser et stocker des données. Il ne permet pas à lui seul d'afficher les données qu'il contient.

Pourtant, XML et HTML sont tous les deux des dérivés d'une langage nommé SGML (Standard Generalized Markup Language). La création d'XML est liée à la complexité de SGML. D'ailleurs, un fichier XML avec sa DTD correspondante peut être traité par un processeur SGML.

XML et java ont en commun la portabilité réalisée grâce à une indépendance vis à vis du système et de leur environnement.

30.2. Les règles pour formater un document XML

Un certain nombre de règles doivent être respectées pour définir un document XML valide et « bien formé ». Pour pouvoir être analysé, un document XML doit avoir une syntaxe correcte. Les principales règles sont :

- le document doit contenir au moins une balise
- chaque balise d'ouverture (exemple <tag>) doit posséder une balise de fermeture (exemple </tag>). Si le tag est vide, c'est à dire qu'il ne possède aucune données (exemple <tag></tag>), un tag abrégé peut être utilisé (exemple correspondant : <tag/>)
- les balises ne peuvent pas être intercalées (exemple <liste><element></liste></element> n'est pas autorisé)
- toutes les balises du document doivent obligatoirement être contenues entre une balise d'ouverture et de fermeture unique dans le document nommée élément racine
- les valeurs des attributs doivent obligatoirement être encadrées avec des quotes simples ou doubles

- les balises sont sensibles à la casse
- Les balises peuvent contenir des attributs même les balises vides
- les données incluses entre les balises ne doivent pas contenir de caractères < et & : il faut utiliser respectivement &lt ; et &amp ;
- La première ligne du document devrait normalement correspondre à la déclaration de document XML : le prologue.

30.3. La DTD (Document Type Definition)

Les balises d'un document XML sont libres. Pour pouvoir valider si le document est correct, il faut définir un document nommé DTD qui est optionnel. Sans sa présence, le document ne peut être validé : on peut simplement vérifier que la syntaxe du document est correcte.

Une DTD est un document qui contient la grammaire définissant le document XML. Elle précise notamment les balises autorisées et comment elles s'imbriquent.

La DTD peut être incluse dans l'en tête du document XML ou être mise dans un fichier indépendant. Dans ce cas, la directive < !DOCTYPE> dans le document XML permet de préciser le fichier qui contient la DTD.

Il est possible d'utiliser une DTD publique ou de définir sa propre DTD si aucune ne correspond à ces besoins.

Pour être valide, un document XML doit avoir une syntaxe correcte et correspondre à la DTD.

30.4. Les parseurs

Il existe plusieurs types de parseur. Les deux plus répandus sont ceux qui utilisent un arbre pour représenter et exploiter le document et ceux qui utilisent des événements. Le parseur peut en plus permettre de valider le document XML.

Ceux qui utilisent un arbre permettent de le parcourir pour obtenir les données et modifier le document. Ceux qui utilisent des événements associent à des événements particuliers des méthodes pour traiter le document.

SAX (Simple API for XML) est une API libre créée par David Megginson qui utilisent les événements pour analyser et exploiter les documents au format XML.

Les parseurs qui produisent des objets composant une arborescence pour représenter le document XML utilisent le modèle DOM (Document Object Model) défini par les recommandations du W3C.

Le choix d'utiliser SAX ou DOM doit tenir compte de leurs points forts et de leurs faiblesses :

	les avantages	les inconvénients
DOM	parcours libre de l'arbre possibilité de modifier la structure et le contenu de l'arbre	gourmand en mémoire doit traiter tout le document avant d'exploiter les résultats
SAX	peut gourmand en ressources mémoire rapide principes faciles à mettre en oeuvre permet de ne traiter que les données utiles	traite les données séquentiellement un peu plus difficile à programmer, il est souvent nécessaire de sauvegarder des informations pour les traiter

SAX et DOM ne fournissent que des définitions : ils ne fournissent pas d'implémentation utilisable. L'implémentation est laissée aux différents éditeurs qui fournissent un parseur compatible avec SAX et/ou DOM.L'avantage d'utiliser l'un deux est que le code utilisé sera compatible avec les autres : le code nécessaire à l'instanciation du parseur est cependant spécifique à chaque fournisseur.

IBM fourni gratuitement un parseur XML : xml4j. Il est téléchargeable à l'adresse suivante : http://www.alphaworks.ibm.com/tech/xml4j

Le groupe Apache développe Xerces à partir de xml4j : il est possible de télécharger la dernière version à l'URL http://xml.apache.org

Sun a développé un projet dénommé Project X. Ce projet a été repris par le groupe Apache sous le nom de Crimson.

Ces trois projets apportent pour la plupart les mêmes fonctionnalitées : ils se distinguent sur des points mineurs : performance, rapidité, facilité d'utilisation etc. ... Ces fonctionnalités évoluent très vite avec les versions de ces parseurs qui se succèdent très rapidement.

Pour les utiliser, il suffit de dézipper le fichier et d'ajouter les fichiers .jar dans la variable définissant le CLASSPATH.

Il existe plusieurs autres parseurs que l'on peut télécharger sur le web.

30.5. L'utilisation de SAX

SAX est l'acronyme de Simple API for XML. Cette API a été développée par David Megginson.

Ce type de parseur utilise des événements pour piloter le traitement d'un fichier XML. Un objet (nommé handler en anglais) doit implémenter des méthodes particulières définies dans une interface de l'API pour fournir les traitements à réaliser : selon les événements, le parseur appelle ces méthodes.

Les dernières informations concernant cette API sont disponible à l'URL : www.megginson.com/SAX/index.html

Les classes de l'API SAX sont regroupées dans le package org.xml.sax

30.5.1. L'utilisation de SAX de type 1

SAX type 1 est composé de deux packages :

- org.xml.sax :
- org.xml.sax.helpers :

SAX définit plusieurs classes et interfaces :

- les interfaces implémentées par le parseur : Parser, AttributeList et Locator
- les interfaces implémentées par le handler : DocumentHandler, ErrorHandler, DTDHandler et EntityHandler
- les classes de SAX :
- des utilitaires rassemblés dans le package org.xml.sax.helpers notamment la classe ParserFactory

Les exemples de cette section utilisent la version 2.0.15 du parseur xml4j d'IBM.

Pour parser un document XML avec un parseur XML SAX de type 1, il faut suivrent les étapes suivantes :

- créer une classe qui implémente l'interface DocumentHandler ou hérite de la classe org.xml.sax.HandlerBase et qui se charge de répondre aux différents événements émis par le parseur
- créer une instance du parseur en utilisant la méthode makeParser() de la classe ParserFactory.
- associer le handler au parseur grâce à la méthode setDocumentHandler()
- exécuter la méthode parse() du parseur

Exemple : avec XML4J

```
import org.xml.sax.*;
import org.xml.sax.helpers.ParserFactory;
import com.ibm.xml.parsers.*;
import java.io.*;

public class MessageXML {
  static final String DONNEES_XML =
  "<?xml version=\"1.0\"?>\n"
  +"<BIBLIOTHEQUE\n>"
  +" <LIVRE>\n"
  +" <TITRE>titre livre 1</TITRE>\n"
  +" <AUTEUR>auteur 1</AUTEUR>\n"
  +" <EDITEUR>editeur 1</EDITEUR>\n"
  +" </LIVRE>\n"
  +" <LIVRE>\n"
  +" <TITRE>titre livre 2</TITRE>\n"
  +" <AUTEUR>auteur 2</AUTEUR>\n"
  +" <EDITEUR>editeur 2</EDITEUR>\n"
  +" </LIVRE>\n"
  +" <LIVRE>\n"
  +" <TITRE>titre livre 3</TITRE>\n"
  +" <AUTEUR>auteur 3</AUTEUR>\n"
  +" <EDITEUR>editeur 3</EDITEUR>\n"
  +" </LIVRE>\n"
  +"</BIBLIOTHEQUE>\n";

  static final String CLASSE_PARSER = "com.ibm.xml.parsers.SAXParser";

  /**
   * Lance l'application.
   * @param args un tableau d'arguments de ligne de commande
   */
  public static void main(java.lang.String[] args) {

    MessageXML m = new MessageXML();
    m.parse();

    System.exit(0);
  }

  public MessageXML() {
    super();
  }

  public void parse() {
    TestXMLHandler handler = new TestXMLHandler();

    System.out.println("Lancement du parseur");

    try {
      Parser parser = ParserFactory.makeParser(CLASSE_PARSER);

      parser.setDocumentHandler(handler);
      parser.setErrorHandler((ErrorHandler) handler);

      parser.parse(new InputSource(new StringReader(DONNEES_XML)));

    } catch (Exception e) {
      System.out.println("Exception capturée : ");
      e.printStackTrace(System.out);
      return;
    }
  }
}
```

Il faut ensuite créer la classe du handler.

Exemple :

```
import java.util.*;

/**
 * Classe utilisee pour gérer les evenement emis par SAX lors du traitement du fichier XML
 */
public class TestXMLHandler extends org.xml.sax.HandlerBase {
        public TestXMLHandler() {
                super();
        }

        /**
         * Actions à réaliser sur les données
         */
        public void characters(char[] caracteres, int debut, int longueur) {
                String donnees = new String(caracteres, debut, longueur);
                System.out.println("   valeur = *" + donnees + "*");
        }

        /**
         * Actions à réaliser lors de la fin du document XML.
         */
        public void endDocument() {
                System.out.println("Fin du document");
        }

        /**
         * Actions à réaliser lors de la détection de la fin d'un element.
         */
        public void endElement(String name) {
                System.out.println("Fin tag " + name);
        }

        /**
         * Actions à réaliser au début du document.
         */
        public void startDocument() {
                System.out.println("Debut du document");
        }

        /**
         * Actions a réaliser lors de la detection d'un nouvel element.
         */
        public void startElement(String name, org.xml.sax.AttributeList atts) {
                System.out.println("debut tag : " + name);
        }
}
```

Resultats :

```
Lancement du parser
Debut du document
debut tag : BIBLIOTHEQUE
   valeur = *
  *
debut tag : LIVRE
   valeur = *
    *
debut tag : TITRE
   valeur = *titre livre 1*
Fin tag TITRE
   valeur = *
    *
debut tag : AUTEUR
   valeur = *auteur 1*
Fin tag AUTEUR
   valeur = *
    *
debut tag : EDITEUR
   valeur = *editeur 1*
Fin tag EDITEUR
```

```
   valeur = *
  *
Fin tag LIVRE
   valeur = *
  *
debut tag : LIVRE
   valeur = *
    *
debut tag : TITRE
   valeur = *titre livre 2*
Fin tag TITRE
   valeur = *
    *
debut tag : AUTEUR
   valeur = *auteur 2*
Fin tag AUTEUR
   valeur = *
    *
debut tag : EDITEUR
   valeur = *editeur 2*
Fin tag EDITEUR
   valeur = *
  *
Fin tag LIVRE
   valeur = *
  *
debut tag : LIVRE
   valeur = *
    *
debut tag : TITRE
   valeur = *titre livre 3*
Fin tag TITRE
   valeur = *
    *
debut tag : AUTEUR
   valeur = *auteur 3*
Fin tag AUTEUR
   valeur = *
    *
debut tag : EDITEUR
   valeur = *editeur 3*
Fin tag EDITEUR
   valeur = *
  *
Fin tag LIVRE
   valeur = *
*
Fin tag BIBLIOTHEQUE
Fin du document
```

Un parseur SAX peut créer plusieurs types d'événements dont les principales méthodes pour y répondre sont :

Evénement	Rôle
startElement()	cette méthode est appelée lors de la détection d'un tag de début
endElement()	cette méthode est appelée lors de la détection d'un tag de fin
characters()	cette méthode est appelée lors de la détection de données entre deux tags
startDocument()	cette méthode est appelée lors du début du traitement du document XML
endDocument()	cette méthode est appelée lors de la fin du traitement du document XML

La classe handler doit redéfinir certaines de ces méthodes selon les besoins des traitements.

En règle générale :

- il faut sauvegarder dans une variable le tag courant dans la méthode startElement()
- traiter les données en fonction du tag courant dans la méthode characters()

La sauvegarde du tag courant est obligatoire car la méthode characters() ne contient pas dans ces paramètres le nom du tag correspondant aux données.

Si les données contenues dans le document XML contiennent plusieurs occurrences qu'il faut gérer avec une collection qui contiendra des objets encapsulant les données, il faut :

- gérer la création d'un objet dans la méthode startElement() lors de la rencontre du tag de début d'un nouvel élément de la liste
- alimenter les attributs de l'objet avec les données de chaque tags utiles dans la méthode characters()
- gérer l'ajout de l'objet à la collection dans la méthode endElement() lors de la rencontre du tag de fin d'élément de la liste

La méthode characters() est appelée lors de la détection de données entre un tag de début et un tag de fin mais aussi entre un tag de fin et le tag début suivant lorsqu'il y a des caractères entre les deux. Ces caractères ne sont pas des données mais des espaces, des tabulations, des retour chariots et certains caractères non visibles.

Pour éviter de traiter les données de ces événements, il y a plusieurs solutions :

- supprimer tous les caractères entre les tags : tous les tags et les données sont rassemblés sur une seule et unique ligne. L'inconvénient de cette méthode est que le message est difficilement lisible par un être humain.
- une autre méthode consiste à remettre à vide la donnée qui contient le tag courant (alimentée dans la méthode startElement()) dans la méthode endElement(). Il suffit alors d'effectuer les traitements dans la méthode characters() uniquement si le tag courant est différent de vide

Exemple :

```
import java.util.*;

/**
 * Classe utilisee pour gérer les evenement emis par SAX lors du traitement du fichier XML
 */
public class TestXMLHandler extends org.xml.sax.HandlerBase {
        private String tagCourant = "";
        public TestXMLHandler() {
                super();
        }

        /**
         * Actions à réaliser sur les données
         */
        public void characters(char[] caracteres, int debut, int longueur) {
                String donnees = new String(caracteres, debut, longueur);
                if (!tagCourant.equals("") {
                        System.out.println("  Element " + tagCourant
             + ", valeur = *" + donnees + "*");
                }
        }

        /**
         * Actions à réaliser lors de la fin du document XML.
         */
        public void endDocument() {
                System.out.println("Fin du document");
        }

        /**
         * Actions à réaliser lors de la détection de la fin d'un element.
         */
        public void endElement(String name) {
                tagCourant = "";
                System.out.println("Fin tag " + name);
        }
```

```
        /**
         * Actions à réaliser au début du document.
         */
        public void startDocument() {
                System.out.println("Debut du document");
        }

        /**
         * Actions a réaliser lors de la detection d'un nouvel element.
         */
        public void startElement(String name, org.xml.sax.AttributeList atts) {
                tagCourant = name;
                System.out.println("debut tag : " + name);
        }
}
```

Résultat :

```
Lancement du parser
Debut du document
debut tag : BIBLIOTHEQUE
   Element BIBLIOTHEQUE, valeur = *
  *
debut tag : LIVRE
   Element LIVRE, valeur = *
    *
debut tag : TITRE
   Element TITRE, valeur = *titre livre 1*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 1*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 1*
Fin tag EDITEUR
Fin tag LIVRE
debut tag : LIVRE
   Element LIVRE, valeur = *
    *
debut tag : TITRE
   Element TITRE, valeur = *titre livre 2*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 2*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 2*
Fin tag EDITEUR
Fin tag LIVRE
debut tag : LIVRE
   Element LIVRE, valeur = *
    *
debut tag : TITRE
   Element TITRE, valeur = *titre livre 3*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 3*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 3*
Fin tag EDITEUR
Fin tag LIVRE
Fin tag BIBLIOTHEQUE
Fin du document
```

- enfin il est possible de vérifier si le premier caractère des données contenues en paramètre de la méthode characters() est un caractère de contrôle ou non grâce à la méthode statique isISOControl() de la classe Character

Exemple :

```
...
/**
 * Actions à réaliser sur les données
 */
public void characters(char[] caracteres, int debut, int longueur) {
        String donnees = new String(caracteres, debut, longueur);

        if (!tagCourant.equals("")) {
                if(!Character.isISOControl(caracteres[debut])) {
                        System.out.println("   Element " + tagCourant
                 +", valeur = *" + donnees + "*");
                }
        }
}
...
```

Résultat :

```
Lancement du parser
Debut du document
debut tag : BIBLIOTHEQUE
debut tag : LIVRE
debut tag : TITRE
   Element TITRE, valeur = *titre livre 1*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 1*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 1*
Fin tag EDITEUR
Fin tag LIVRE
debut tag : LIVRE
debut tag : TITRE
   Element TITRE, valeur = *titre livre 2*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 2*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 2*
Fin tag EDITEUR
Fin tag LIVRE
debut tag : LIVRE
debut tag : TITRE
   Element TITRE, valeur = *titre livre 3*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 3*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 3*
Fin tag EDITEUR
Fin tag LIVRE
Fin tag BIBLIOTHEQUE
Fin du document
```

SAX définit une exception de type SAXParserException lorsque le parseur contient une erreur dans le document en cours de traitement. Les méthodes getLineNumber() et getColumnNumber() permettent d'obtenir la ligne et la colonne ou l'erreur a été détectée.

Exemple :

```
try {
...
```

```
}  catch (SAXParseException e) {
  System.out.println("Erreur lors du traitement du document XML");
  System.out.println(e.getMessage());
  System.out.println("ligne : "+e.getLineNumber());
  System.out.println("colonne : "+e.getColumnNumber());
  }
```

Pour les autres erreurs, SAX définit l'exception SAXException.

30.5.2. L'utilisation de SAX de type 2

SAX de type 2 apporte principalement le support des espaces de noms. Les classes et les interfaces sont toujours définies dans les packages org.xml.sax et ses sous packages.

SAX de type 2 définit quatre interfaces que l'objet handler doit ou peut implémenter :

- ContentHandler : interface qui définit les méthodes appellées lors du traitement du document
- ErrorHandler : interface qui définit les méthodes appellées lors du traitement des warnings et de erreurs
- DTDHandler : interface qui définit les méthodes appellées lors du traitement de la DTD
- EntityResolver

Plusieurs classes et interfaces de SAX de type 1 sont deprecated :

	ancienne entité SAX 1	nouvelle entité SAX 2
Interface	org.xml.sax.Parser	XMLReader
	org.xml.sax.DocumentHandler	ContentHandler
	org.xml.sax.AttributeList	Attributes
Classes	org.xml.sax.helpers.ParserFactory	
	org.xml.sax.HandlerBase	DefaultHandler
	org.xml.sax.helpers.AttributeListImpl	AttributesImpl

Les principes de fonctionnement de SAX 2 sont très proche de SAX 1.

Exemple :

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class TestSAX2
{

  public static void main(String[] args)
  {
    try
    {
      Class c = Class.forName("org.apache.xerces.parsers.SAXParser");
      XMLReader reader = (XMLReader)c.newInstance();
      TestSAX2Handler handler = new TestSAX2Handler();
      reader.setContentHandler(handler);
      reader.parse("test.xml");
    }
    catch(Exception e){System.out.println(e);}
  }
```

```
}


class TestSAX2Handler extends DefaultHandler
{
  private String tagCourant = "";

  /**
   * Actions a réaliser lors de la detection d'un nouvel element.
   */
  public void startElement(String nameSpace, String localName,
    String qName, Attributes attr) throws SAXException  {
    tagCourant = localName;
    System.out.println("debut tag : " + localName);
  }

  /**
   * Actions à réaliser lors de la détection de la fin d'un element.
   */
  public void endElement(String nameSpace, String localName,
    String qName) throws SAXException {
    tagCourant = "";
    System.out.println("Fin tag " + localName);
  }

  /**
   * Actions à réaliser au début du document.
   */
  public void startDocument() {
    System.out.println("Debut du document");
  }

  /**
   * Actions à réaliser lors de la fin du document XML.
   */
  public void endDocument() {
    System.out.println("Fin du document");
  }

  /**
   * Actions à réaliser sur les données
   */
  public void characters(char[] caracteres, int debut,
    int longueur) throws SAXException {
    String donnees = new String(caracteres, debut, longueur);

    if (!tagCourant.equals("")) {
      if(!Character.isISOControl(caracteres[debut])) {
        System.out.println("   Element " + tagCourant +",
          valeur = *" + donnees + "*");
      }
    }
  }
}
```

30.6. L'utilisation de DOM

DOM (Document Objet Model) est un autre type de parseur défini par une recommandation du consortium W3.

Ce modèle propose de parcourir tout le document XML pour construire une arborescence composée de noeuds en mémoire. Une fois cette arbre construit, il est possible de le parcourir mais aussi de le modifier.

Cette section est en cours d'écriture

30.7. La génération de données au format XML

Il existe plusieurs façon de générer des données au format XML :

- coder cette génération à la main en écrivant dans un flux

Exemple :

```
public void service(HttpServletRequest request, HttpServletResponse response)
  throws IOException, ServletException {
  response.setContentType("text/xml");
  PrintWriter out = response.getWriter();

  out.println("<?xml version=\"1.0\"?>");
  out.println("<BIBLIOTHEQUE>");
  out.println(" <LIVRE>");
  out.println(" <TITRE>titre livre 1</TITRE>");
  out.println(" <AUTEUR>auteur 1</AUTEUR>");
  out.println(" <EDITEUR>editeur 1</EDITEUR>");
  out.println(" </LIVRE>");
  out.println(" <LIVRE>");
  out.println(" <TITRE>titre livre 2</TITRE>" );
  out.println(" <AUTEUR>auteur 2</AUTEUR>");
  out.println(" <EDITEUR>editeur 2</EDITEUR>" );
  out.println(" </LIVRE>");
  out.println(" <LIVRE>");
  out.println(" <TITRE>titre livre 3</TITRE>");
  out.println(" <AUTEUR>auteur 3</AUTEUR>");
  out.println(" <EDITEUR>editeur 3</EDITEUR>");
  out.println(" </LIVRE>");
  out.println("</BIBLIOTHEQUE>");
  }
}
```

- utiliser JDOM pour construire le document et le sauvegarder

Cette section est en cours d'écriture

30.8. JAXP : Java API for XML Parsing

JAXP est une API développée par Sun qui ne fournit pas une nouvelle méthode pour parser un document XML mais propose une interface commune pour appeler et paramétrerun parseur de façon indépendante de tout fournisseur et normaliser la source XML à traiter. En utilisant un code qui respecte JAXP, il est possible d'utiliser n'importe quel parseur qui répond à cette API tel que Crimson le parseur de Sun ou Xerces le parseur du groupe Apache.

JAXP supporte pour le moment les parseurs de type SAX et DOM.

	JAXP 1.0	JAXP 1.1
SAX	type 1	type 2
DOM	niveau 1	niveau 2

Par exemple, sans utiliser JAXP, il existe deux méthodes pour instancier un parseur de type SAX :

- créer une instance de la classe de type SaxParser
- utiliser la classe ParserFactory qui demande en paramètre le nom de la classe de type SaxParser

Ces deux possibilités nécessitent une recompilation d'une partie du code lors du changement du parseur.

JAXP propose de fournir le nom de la classe du parseur en paramètre à la JVM sous la forme d'une propriété système. Il n'est ainsi plus nécessaire de procéder à une recompilation mais simplement de mettre jour cette propriété et le CLASSPATH pour qu'il référence les classes du nouveau parseur.

Le parseur de Sun et les principaux parseurs XML en Java implantent cette API et il est très probable que tous les autres fournisseurs suivent cet exemple.

30.8.1. JAXP 1.1

JAXP version 1.1 contient une documentation au format javadoc, des exemples et trois fichiers jar :

- jaxp.jar : contient l'API JAXP
- crimson.jar : contient le parseur de Sun
- xalan.jar : contient l'outils du groupe apache pour les transformations XSL

L'API JAXP est fournie avec une implémentation de référence de deux parseurs (une de type SAX et une de type DOM) dans le package org.apache.crimson et ses sous packages.

JAXP ce compose de plusieurs packages :

- javax.xml.parsers
- javax.xml.transform
- org.w3c.dom
- org.xml.sax

JAXP définit deux exceptions particulières :

- FactoryConfigurationError est levée si la classe du parseur précisée dans la variable System ne peut être instanciée
- ParserConfigurationException est levée lorsque les options précisées dans la factory ne sont pas supportées par le parseur

30.8.2. L'utilisation de JAXP avec un parseur de type SAX

L'API JAXP fournie la classe abstraite SAXParserFactory qui fournie une méthode pour récupérer une instance d'un parseur de type SAX grace à une de ces méthodes. Une classe fille de la classe SAXParserFactory permet d'être instanciée.

La propriété système javax.xml.parsers.SAXParserFActory permet de préciser la classe fille qui hérite de la classe SAXParserFactory et qui sera instanciée.

Remarque : cette classe n'est pas thread safe.

La méthode statique newInstance() permet d'obtenir une instance de la classe SAXParserFactory : elle peut lever une exception de type FactoryConfigurationError.

Avant d'obtenir une instance du parseur, il est possible de fournir quelques paramètres à la Factory pour lui permettre de configurer le parseur.

La méthode newSAXParser() permet d'obtenir une instance du parseur de type SAXParser : peut lever une exception de type ParserConfigurationException.

Les principales méthodes sont :

Méthode	Rôle
boolean isNamespaceAware()	indique si la factory est configurée pour instancier des parseurs qui prennent en charge les espaces de noms
boolean isValidating()	indique si la factory est configurée pour instancier des parseurs qui valide le document XML lors de son traitement
static SAXParserFactory newInstance()	permet d'obtenir une instance de la factory
SAXParser newSAXParser()	permet d'obtenir une nouvelle instance du parseur de type SAX configuré avec les options fournie à la factory
setNamespaceAware(boolean)	configure la factory pour instancier un parseur qui prend en charge les espaces de noms ou non selon le paramètre fourni
setValidating(boolean)	configure la factory pour instancier un parseur qui valide le document XML lors de son traitement ou non selon le paramètre fourni

Exemple :

```
SAXParserFActory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
parser.parse(new File(args[0), new handler());
```

30.9. XSLT (Extensible Stylesheet Language Transformations)

XSLT est une recommandation du consortium W3C qui permet de transformer facilement des documents XML en d'autres documents standards sans programmation. Le principe est de définir une feuille de style qui indique comment transformer le document XML et de le fournir avec le document à un processeur XSLT.

On peut produire des documents de différents formats : XML, HTML, XHTML, WML, PDF, etc...

XSLT fait parti de XSL avec les recommandations :

- XSL−FO : flow object
- XPath : langage pour spécifier un élement dans un document. Ce langage est utilisé par XSL.

Une feuille de style XSLT est un fichier au format XML qui contient les informations nécessaires au processeur pour effectuer la transformation.

Le composant principal d'une feuille de style XSLT est le template qui définit le moyen de transformer un élément du document XML dans le nouveau document.

XSLT est très relativement complet et complexe : cette section n'est qu'une présentation rapide de quelques fonctionnalités de XSLT. XSLT possède plusieurs fonctionnalités avancées, tel que la sélection des élément à traiter, filter des éléments ou trier les éléments.

30.9.1. XPath

XML Path ou XPath est une spécification qui fournit une syntaxe pour permettre de sélectionner un ou plusieurs éléments dans un document XML. Il existe sept types d'éléments différents :

- racine (root)
- element
- text
- attribute (attribute)
- commentaire (comment)
- instruction de traitement (processing instruction)
- espace de nommage (name space)

Cette section ne présente que les fonctionnalités de base de XPath.

XPath est utilisé dans plusieurs technologies liées à XML tel que XPointer et XSLT.

Un document XMLpeut être représenté sous la forme d'un arbre composé de noeud. XPath grace à une notation particulière permet localisation précisement un composant de l'arbre.

La notation reprend une partie de la notation utiliser pour naviguer dans un système d'exploitation, ainsi :

- le séparateur est le le slash /
- pour préciser un chemin à partir de la racine (chemin absolue), il faut qu'il commence par un /
- un double point .. permet de préciser l'élément père de l'élément courant
- un simple point . permet de préciser l'élément courant
- un arobase @ permet de préciser un attribut d'un élément
- pour préciser l'indice d'un élément il faut le préciser entre crochets

XPath permet de filtrer les élements sur différents critères en plus de leur nom

- @categorie="test" : recherche un attribut dont le nom est categorie est dont la valeur est "test"
- une barre verticale | permet de preciser deux valeurs

30.9.2. La syntaxe de XSLT

Une feuille de style XSLT est un document au format XML. Il doit donc respecter toute les règles d'un tel document. Pour préciser les différentes instructions permettant de réaliser la transformation, un espace de nommage particulier est utilisé : xsl. Tout les tags de XSLT commencent donc par ce préfixe, ainsi le tag racine du document est xsl:stylesheet. Ce tag racine doit obligatoirement posséder un attribut version qui précise la version de XSLT utilisé.

Exemple : une feuille de style minimale qui ne fait rien

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

</xsl:stylesheet>
```

Le tag xsl:output permet de préciser le format de sortie. Ce tag possède plusieurs attributs :

- method : cet attribut permet de préciser le format. Les valeurs possiblent sont : texte, xml ou html
- indent : cet attribut permet de définir si la sortie doit être indentée ou non. Les valeurs possiblent sont : yes ou non
- encoding : cet aatribut permet de préciser le jeux de caractères utilisé pour la sortie

Pour effectuer la transformation, le document doit contenir des règles. Ces règles suivent une syntaxe particulière et sont

contenues dans des modèles (templates) associés à un ou plusieurs éléments désignés avec un motif au format XPath.

Un modèle est défini grace au tag xsl:template. La valeur de l'attribut match permet de fournir le motif au format XPath qui sélectionnera le ou les élements sur lequel le modèle va agir.

Le tag xsl:apply−templates permet de demander le traitements des autres modèles définis pour chacun des noeuds fils du noeud courant.

Le tag xsl:value−of permet d'extraire la valeur de l'élément respectant le motif XPath fourni avec l'attribut select.

Il existe beaucoup d'autre tags notamment plusieurs qui permettent d'utiliser de structures de contrôles de type itératifs ou conditionnels.

Le tag xsl:for−each permet parcourir un ensemble d'élement sélectionner par l'attribut select. Le modèle sera appliqué sur chacun des éléments de la liste

Le tag xsl:if permet d'éxécuter le modèle si la condition précisée par l'attribut test au format XPath est juste. XSLT ne défini pas de tag équivalent à la partie else : il faut définir un autre tag xsl:if avec une condition opposée.

Le tag xsl:choose permet de définir plusieurs conditions. Chaque condition est précisée grace à l'attribut xsl:when avec l'attribut test. Le tag xsl:otherwise permet de définir un cas par défaut qui ne correspond aux autres cas défini dans le tag xsl:choose.

Le tag xsl:sort permet de trier un ensemble d'éléments. L'attribut select permet de préciser les élément qui doivent être triés. L'attribut data−type permet de préciser le format des données (text ou number). l'attribut order permet de préciser l'ordre de tri (ascending ou descending).

30.9.3. Exemple avec Internet Explorer

Le plus simple pour tester une feuille XSLT qui génère une page HTML est de la tester avec Internet Explorer version 6. Cette version est entièrement compatible avec XML et XSLT. Les versions 5 et 5.5 ne sont que partiellement compatible. Les versions antérieures ne le sont pas du tout.

30.9.4. Exemple avec Xalan 2

Xalan 2 utilise l'API JAXP.

Exemple : TestXSL2.java

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import org.xml.sax.*;
import java.io.IOException;

public class TestXSL2
{
  public static void main(String[] args)
        throws TransformerException, TransformerConfigurationException,
         SAXException, IOException
        {

    TransformerFactory tFactory = TransformerFactory.newInstance();
    Transformer transformer = tFactory.newTransformer(new StreamSource("test.xsl"));

    transformer.transform(new StreamSource("test.xml"), new StreamResult("test.htm"));
  }
}
```

Exemple : la feuille de style XSL test.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns="http://www.w3.org/TR/REC-html40">

<xsl:output method="html" indent="no" />
<xsl:strip-space elements="*" />

<xsl:template match="/">
<HTML>
<HEAD>
<TITLE>Test avec XSL</TITLE>
</HEAD>
<xsl:apply-templates />
</HTML>
</xsl:template>

<xsl:template match="BIBLIOTHEQUE">
<BODY>
<H1>Liste des livres</H1>
<TABLE border="1" cellpadding="4">
<TR><TD>Titre</TD><TD>Auteur</TD><TD>Editeur</TD></TR>
<xsl:apply-templates />
</TABLE>
</BODY>
</xsl:template>

<xsl:template match="LIVRE">
<TR>
<TD><xsl:apply-templates select="TITRE"/></TD>
<TD><xsl:apply-templates select="AUTEUR"/></TD>
<TD><xsl:apply-templates select="EDITEUR"/></TD>
</TR>
</xsl:template>
</xsl:stylesheet>
```

Exemple : compilation et execution avec Xalan

```
javac TestXSL2.java -classpath .;xalan2.jar
java -cp .;xalan2.jar TestXSL2
```

Cette section est en cours d'écriture

30.10. Les modèles de document

Devant les faibles possibilités de manipulation d'un document XML avec SAX et le manque d'intégration à java de DOM (qui n'a pas été écrit pour java), plusieurs projets se sont développés pour proposer des modèles de représentation des documents XML qui utilise un ensemble d'api se basant sur celles existant en java.

Les deux projets les plus connus sont JDOM et Dom4J. Bien que différent, ces deux projets ont de nombreux points communs :

- ils ont le même but
- ils sont écrits en java et ne s'utilisent qu'en java
- ils utilisent soit SAX soit DOM pour créer le modèle de document en mémoire
- ils utilisent l'api Collection de Java2.

30.11. JDOM

Malgrès la similitude de nom entre JDOM et DOM, ces deux API sont très différentes. JDOM est une API uniquement java car elle s'appuie sur un ensemble de classe de java notamment les collections. Le but de JDOM n'est pas de définir un nouveau type de parseur mais de faciliter la manipulation au sens large de document XML : lecture d'un document, représentation sous forme d'arborescence, manipulation de cette arbre, définition d'un nouveau document, exportation vers plusieurs cibles ... Dans le rôle de manipulation sous forme d'arbre, JDOM possède moins de fonctionnalité que DOM mais en contre partie il offre une plus grande facilité pour répondre au cas les plus classique d'utilisation.

Pour parser un document XML, JDOM utilise un parseur externe de type SAX ou DOM.

JDOM est une Java Specification Request numéro 102 (JSR−102).

30.11.1. Installation de JDOM sous Windows

Pour utiliser JDOM il faut construire la bibliothèque grace à l'outils ant. Ant doit donc être installé sur la machine.

Il faut aussi que la variable JAVA_HOME soit positionnée avec le répertoire qui contient le JDK.

set JAVA_HOME=c:\j2sdk1.4.0−rc

Il suffit alors simplement d'executer le fichier build.bat situé dans le répertoire d'installation de jdom.

Un message informe de la fin de la compilation :

package:
[jar] Building jar: C:\java\jdom−b7\build\jdom.jar

BUILD SUCCESSFUL

Total time: 1 minutes 33 seconds

Le fichier jdom.jar est créé dans le répertoire build.

Pour utiliser JDOM dans un projet, il faut obligatoirement avoir un parseur XML SAX et/ou DOM. Pour les exemples de cette section j'ai utilisé xerces. Il faut aussi avoir le fichier jaxp.jar.

Pour compiler et éxecuter les exemples de cette section, j'ai utilisé le script suivant :

javac %1.java −classpath .;jdom.jar;xerces.jar;jaxp.jar

java −classpath .;jdom.jar;xerces.jar;jaxp.jar %1

30.11.2. Les différentes entités de JDOM

Pour traiter un document XML, JDOM définit plusieurs entités qui peuvent être regroupées en trois groupes :

- les éléments de l'arbre
    - le document : la classe Document
    - les éléments : la classe Element
    - les commentaires : la classe Comment
    - les attributs : la classe Attribute
- les entités pour obtenir un parseur :
    - les classe SAXBuilder et DOMBuilder
- les entités pour produire un document

- ♦ les classes XMLOutputter, SAXOutputter, DOMOutputter

Ces classes sont regroupées dans cinq packages :

- org.jdom
- org.jdom.adapters
- org.jdom.input
- org.jdom.output
- org.jdom.transform

Attention : cette API a énormément évoluée jusqu'à sa version 1.0. Beaucoup de méthode ont été déclarée deprecated.

30.11.3. La classe Document

La classe org.jdom.Document encapsule l'arbre dans le lequel JDOM stocke le document XML.Pour obtenir un objet Document, il y a deux possibilités :

- utiliser un objet XXXBuilder qui va parser un document XML existant et créer l'objet Document en utilisant un parseur
- instancier un nouvel objet Document pour créer un nouveau document XML

Pour créer un nouveau document, il suffit d'instancier un objet Document en utilisant un des constructeurs fournis dont les principaux sont :

Constructeur	Rôle
Document()	
Document(Element)	Création d'un document avec l'élement racine fourni
Document(Element, DocType)	Création d'un document avec l'élement racine et la déclaration doctype fourni

Exemple (code java 1.2) :

```
import org.jdom.*;

public class TestJDOM2 {
        public static void main(String[] args) {
                Element racine = new Element("bibliothèque");
                Document document = new Document(racine);

        }
}
```

Pour obtenir un document à partir d'un document XML existant, JDOM propose deux classes regroupées dans le package org.jdom.input qui implémentent l'interface Builder. Cette interface définit la méthode build() qui renvoie un objet de type Document et qui est surchargée pour utiliser trois sources différentes : InputStream, File et URL. Les deux classes sont SAXBuilder et DOMbuilder.

La classe SAXBuilder permet de parser le document XML avec un parseur de type SAX compatible JAXP, de créer un arbre JDOM et renvoie un objet de type Document. SAXBuilder possède plusieurs constructeurs qui permettent de préciser la classe du parseur à utiliser et/ou un boolean qui indique si le document doit être validé.

Exemple (code java 1.2) :

```
import org.jdom.*;
import org.jdom.input.*;
import java.io.*;

public class TestJDOM3 {
        public static void main(String[] args) {
                try {
                        SAXBuilder builder = new SAXBuilder();
                        Document document = builder.build(new File("test.xml"));
                } catch(JDOMException e) {
                        e.printStackTrace();
                }

        }
}
```

La classe Document possède plusieurs méthodes dont les principales sont :

Méthode	Rôle
Document addContent(Comment)	Ajouter un commentaire au document
List getContent()	Renvoie un objet List qui contient chaque élément du document
DocType getDocType()	Renvoie un objet contenant les caractéristiques doctype du document
Element getRootElement()	Renvoie l'élément racine du document
Document setRootElement(Element)	Défini l'élément racine du document

La classe DocType encapsule les données sur le type du document.

30.11.4. La classe Element

La classe Element représente un élément du document. Un élément peut contenir du texte, des attributs, des commentaires, et tous les autres éléments définis par la norme XML. Cette classe possède plusieurs constructeurs dont les principaux sont :

Constructeur	Rôle
Element()	Créer un objet par défaut
Element(String)	Créer un objet, en précisant son nom
Element(String, String)	Créer un objet, en précisant son nom et son espace de nommage

La classe Element possède de nombreuses méthodes pour obtenir, ajouter ou supprimer une entité de l'élément (une élement enfant, un texte, un attribut, un commentaire ...).

Méthode	Rôle
Element addContent(Comment)	Ajouter un commentaire à l'élément
Element addContent(Element)	Ajouter un élément fils à l'élément
Element addContent(String text)	Ajouter des données sous forme de texte à l'élément

Attribute getAttribute(String)	Renvoie l'attribut dont le nom est fourni en paramètre
Attribute getAttribute(String, Namespace)	Renvoie l'attribut dont le nom et l'espace de nommage sont fourni en paramètre
List getAttributes()	Renvoie une liste qui contient tous les attributs
String getAttributeValue(String)	Renvoie la valeur de l'attribut dont le nom est fourni en paramètre
String getAttributeValue(String, Namespace)	Renvoie la valeur de l'attribut dont le nom et l'espace de nommage sont fourni en paramètre
Element getChild(String)	Renvoie le premier élément enfant dont le nom est fourni en paramètre
Element getChild(String, Namespace)	Renvoie le premier élément enfant dont le nom et l'espace de nommage sont fourni en paramètre
List getChildren()	Renvoie une liste qui contient tous les élements enfants directement rattachés à l'élément
List getChildren(String)	Renvoie une liste qui contient tous les élements enfants directement rattachés à l'élément dont le nom est fourni en paramètre
List getChildren(String, Namespace)	Renvoie une liste qui contient tous les élements enfants directement rattachés à l'élément dont le nom et l'espace de nommage est fourni en paramètre
String getChildText(String name)	Renvoie le texte du premier élément enfant dont le nom est fourni en paramètre
String getChildText(String, Namespace)	Renvoie le texte du premier élément enfant dont le nom et l'espace de nommage sont fourni en paramètre
List getContent()	Renvoie une liste qui contient contient toutes les entités de l'élément (texte, element, commentaire ...)
Document getDocument()	Renvoie l'objet Document qui contient l'élément
Element getParent()	Renvoie l'élément père de l'élément
String getText()	Renvoie les données au format texte contenues dans l'élément
boolean hasChildren()	Renvoie un élément qui indique si l'lément possède des élements fils
boolean isRootElement()	Renvoie un boolen qui indique si l'élément est l'élément racine du document
boolean removeAttribute(String)	Supprime l'attribut dont le nom est fourni en parametre
boolean removeAttribute(String, Namespace)	Supprime l'attribut dont le nom et l'espace de nommage sont fournis en paramètre
boolean removeChild(String)	Supprime l'élément enfant dont le nom est fourni en paramètre
boolean removeChild(String, Namespace)	Supprime l'élément enfant dont le nom et l'espace de nommage sont fournis en paramètre
boolean removeChildren()	Supprime tous les éléments enfants
boolean removeChildren(String)	Supprime tous les éléments enfants dont le nome est fourni en paramètre
boolean removeChildren(String, Namespace)	Supprime tous les éléments enfants dont le nom et l'espace de nommage sont fournis en paramètre
boolean removeContent(Comment)	Supprime le commentaire fourni en paramètre
boolean removeContent(Element)	Supprime l'élément fourni en paramètre

Element setAttribute(Attribute)	Ajoute un attribut
Element setAttribute(String, String)	Ajoute un attribut dont le nom et la valeur sont fournis en paramètre
Element setAttribute(String, String, Namespace)	Ajoute un attribut dont le nom, la valeur et l'espace de nommage sont fournis en paramètre
Element setText(String)	Mettre à jour les données au format texte de l'élément

Pour obtenir l'élément racine d'un document, il faut utiliser la méthode getRootElement() de la classe Document. Celle ci renvoie un objet de type Element. Il est ainsi possible d'utiliser les méthodes ci dessous pour parcourir et modifier le contenu du document.

L'utilisation de la classe Elément est très facile.

Pour créer un élément, il suffit d'instancier un objet de type Element.

Exemple (code java 1.2) :

```
          Element element = new Element("element1");
          element.setAttribute("attribut1","valeur1");
          element.setAttribute("attribut2","valeur2");
```

La classe possède plusieurs méthodes pour obtenir les entités de l'élément, un élément fils particulier ou une liste d'élément fils. Un appel successif à ces méthodes permet d'obtenir un élément précis du document.

Exemple (code java 1.2) :

```
import org.jdom.*;
import org.jdom.input.*;
import java.io.*;
import java.util.*;

public class TestJDOM5 {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("test.xml"));

      Element element = document.getRootElement();
      List livres=element.getChildren("LIVRE");
      ListIterator iterator = livres.listIterator();
      while (iterator.hasNext()) {
        Element el = (Element) iterator.next();
        System.out.println("titre = "+el.getChild("TITRE").getText());
      }
    } catch(JDOMException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat:

```
titre = titre livre 1
titre = titre livre 2
titre = titre livre 3
```

L'inconvénient d'utiliser un nom de tag pour rechecher un élément est qu'il faut être sur que l'élément existe sinon une exception de type nullPointerException est levée. Le plus simple est d'avoir une DTD et d'activer la validation du document pour le parseur.

30.11.5. La classe Comment

Cette section est en cours d'écriture

30.11.6. La classe Namespace

JDOM gère les espace de nom grace à la classe Namespace. Un espace de nom est composé d'un préfix auquel on associé une URI. JDOM gére les espaces de nom de lui même. La méthode statique getNamespace() permet de retrouver ou de créer un espace de nom.

30.11.7. La classe Attribut

Cette section est en cours d'écriture

30.11.8. La sortie de document

JDOM prévoit plusieurs classes pour permettre d'exporte le document contenu dans un objet de type Document.

Le plus utilisé est de type XMLOutputter qui permet d'envoyer le document XML dans un flux. Il est possible de fournir plusieurs paramètres pour formatter la sortie du document. Cette classe possède plusieurs constructeurs dont les principaux sont :

Constructeur	Rôle
XMLOutputter()	Créer un objet par défaut, sans paramètre de formattage
XMLOutputter(String)	Créer un objet, en précisant une chaine pour l'indentation
XMLOutputter(String, boolean)	Créer un objet, en précisant une chaine pour l'indentation et un boolean qui indique si une nouvelle ligne doit être ajoutée après chaque élément
XMLOutputter(String, boolean, String)	Créer un objet, en précisant une chaine pour l'indentation, un boolean qui indique si une nouvelle ligne doit être ajoutée après chaque élément et une chaine qui précise le jeux de caractères à utiliser pour formatter le document

```
XMLOutputter outputter = new XMLOutputter();
outputter.output(doc, System.out);
```

Si le fichier XML n'a besoin d'être exploité que par un programme informatique, il est préférable de supprimer toute forme de formattage.

```
XMLOutputter outputter = new XMLOutputter("", false);
outputter.output(doc, System.out);
```

Cette section est en cours d'écriture

30.12. dom4j

dom4j est un framework open source pour manipuler des données XML, XSL et Xpath. Il est entièrement développé en Java et pour Java.

Dom4j n'est pas un parser mais propose un modèle de représentation d'un document XML et une API pour en faciliter l'utilisation. Pour obtenir une tel représentation, dom4j utilise soit SAX, soit DOM. Comme il est compatible JAXP, il est possible d'utiliser toute implémentation de parser qui implémente cette API.

La version de dom4j utilisée dans cette section est la 1.3

30.12.1. Installation de dom4j

Download de la dernière version à l'url http://dom4j.org/download.html

Il suffit de dezipper le fichier downloadé. Celui ci contient de nombreuses bibliothèques (ant, xalan, xerces, crimson, junit, ...), le code source du projet, et la documentation.

Le plus simple pour utiliser rapidement dom4j est de copier les fichiers jar contenu dans le répertoire lib dans le répertoire ext du répertoire %JAVA_HOME%/jre/lib/ext ainsi que les fichier dom4j.jar et dom4j−full.jar.

30.12.2. La création d'un document

Dom4j encapsule un document dans un objet de type org.dom4j.Document. Dom4j propose des API pour facilement créer un tel objet qui va être le point d'entrée de la représentation d'un document XML .

Exemple utilisant SAX :

```
import org.dom4j.*;
import org.dom4j.io.*;
public class Testdom4j_1 {
  public static void main(String args[]) {
    DOCUMENT DOCUMENT;
    try {
      SAXReader xmlReader = new SAXReader();
      document = xmlReader.read("test.xml");
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Pour éxecuter ce code, il suffit d'executer

java −cp .;%JAVA_HOME%/jre/lib/ext/dom4j−full.jar Testdom4j_1.bat

Exemple à partir d'une chaine de caractères :

```
import org.dom4j.*;
public class Testdom4j_8 {
  public static void main(String args[]) {
    Document document = null;
    String texte = "<bibliotheque><livre><titre>titre 1</titre><auteur>auteur 1</auteur>"
         +"<editeur>editeur 1</editeur></livre></bibliotheque>";
    try {
      document = DocumentHelper.parseText(texte);
    } catch (Exception e){
     e.printStackTrace();
    }
  }
}
```

30.12.3. Le parcours d'un document

Le parcours du document construit peut se faire de plusieurs façon :

- utilisation de l'API collection
- utilisation de XPath
- utilisation du pattern Visitor

Le parcours peut se faire en utilisant l'API collection de Java.

Exemple : obtenir tous les noeuds fils du noeud racine

```
import org.dom4j.*;
import org.dom4j.io.*;
import java.util.*;
public class Testdom4j_2 {
  public static void main(String args[]) {
    Document document;
    org.dom4j.Element racine;
    try {
      SAXReader xmlReader = new SAXReader();
      document = xmlReader.read("test.xml");
      racine = document.getRootElement();
      Iterator it = racine.elementIterator();
      while(it.hasNext()){
        Element element = (Element)it.next();
        System.out.println(element.getName());
      }
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Un des grands intérêts de dom4j est de proposé une recherche dans le document en utilisant la technologie Xpath.

Exemple : obtenir tous les noeuds fils du noeud racine

```
import org.dom4j.*;
import org.dom4j.io.*;
import java.util.*;
public class Testdom4j_3 {
```

```
  public static void main(String args[]) {
    Document document;
    try {
      SAXReader xmlReader = new SAXReader();
      document = xmlReader.read("test.xml");
      XPath xpathSelector = DocumentHelper.createXPath("/bibliotheque/livre/auteur");
      List liste = xpathSelector.selectNodes(document);
      for ( Iterator it = liste.iterator(); it.hasNext(); ) {
        Element element = (Element) it.next();
        System.out.println(element.getName()+" : "+element.getText());
      }
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

30.12.4.La modification d'un document XML

L'interface Document propose plusieurs méthodes pour modifier la structure du document.

Méthode	Rôle
Document addComment(String)	Ajouter un commentaire
void setDocType(DocumentType)	Modifier les caractéristique du type de document
void setRootElement(Element)	Modifier l'élement racine du document

L'interface Element propose plusieurs méthodes pour modifier un élément du document.

Méthode	Rôle
void add(…)	Méthode surchargée qui permet d'ajouter un attribut, une entité, un espace nommage ou du texte à l'élément
Element addAttribute(String, String)	Ajouter un attribut à l'élément
Element addComment(String)	Ajouter un commentaire à l'élément
Element addEntity(String, String)	Ajouter une entité à l'élément
Element addNameSpace(String, String)	Ajouter un espace de nommage à l'élément
Element addText(String)	Ajouter un text à l'élément

30.12.5. La création d'un nouveau document XML

Il est très facile de créer un document XML

La classe DocumentHelper propose une méthode createDocument() qui renvoie une nouvelle instance de la classe Document. Il suffit alors d'ajouter chacun des nœuds de l'arbre du nouveau document XML en utilisant la méthode addElement() de la classe Document.

Exemple :

```
import org.dom4j.*;
public class Testdom4j_4 {
  public static void main(String args[]) {
    Document document = DocumentHelper.createDocument();
    Element root = document.addElement( "bibliotheque" );
    Element livre = null;
    try {
    livre = root.addElement("livre");
    livre.addElement("titre").addText("titre 1");
    livre.addElement("auteur").addText("auteur 1");
    livre.addElement("editeur").addText("editeur 1");
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

30.12.6. Exporter le document

Pour écrire le document XML dans un fichier, une méthode de la classe Document permet de réaliser cette action très simplement

Exemple :

```
import org.dom4j.*;
import java.io.*;
public class Testdom4j_5 {
  public static void main(String args[]) {
    Document document = DocumentHelper.createDocument();
    Element root = document.addElement( "bibliotheque" );
    Element livre = null;
    try {
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 1");
      livre.addElement("auteur").addText("auteur 1");
      livre.addElement("editeur").addText("editeur 1");
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 2");
      livre.addElement("auteur").addText("auteur 2");
      livre.addElement("editeur").addText("editeur 2");
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 3");
      livre.addElement("auteur").addText("auteur 3");
      livre.addElement("editeur").addText("editeur 3");
      FileWriter out = new FileWriter( "test2.xml" );
      document.write( out );
      out.close();
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Pour pouvoir agir sur le formattage du document ou pour utiliser un flux différent, il faut utiliser la classe XMLWriter

Exemple :

```
import org.dom4j.*;
import org.dom4j.io.*;
import java.io.*;
public class Testdom4j_6 {
  public static void main(String args[]) {
    Document document = DocumentHelper.createDocument();
    Element root = document.addElement( "bibliotheque" );
    Element livre = null;
    try {
      livre = root.addElement("livre");
```

```
      livre.addElement("titre").addText("titre 1");
      livre.addElement("auteur").addText("auteur 1");
      livre.addElement("editeur").addText("editeur 1");
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 2");
      livre.addElement("auteur").addText("auteur 2");
      livre.addElement("editeur").addText("editeur 2");
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 3");
      livre.addElement("auteur").addText("auteur 3");
      livre.addElement("editeur").addText("editeur 3");
      OutputFormat format = OutputFormat.createPrettyPrint();
      XMLWriter writer = new XMLWriter( System.out, format );
      writer.write( document );
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
C:\test_dom4j>java -cp .;c:\j2sdk1.4.0_01/jre/lib/ext/dom4j
-full.jar Testdom4j_6
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <titre>titre 1</titre>
    <auteur>auteur 1</auteur>
    <editeur>editeur 1</editeur>
  </livre>
  <livre>
    <titre>titre 2</titre>
    <auteur>auteur 2</auteur>
    <editeur>editeur 2</editeur>
  </livre>
  <livre>
    <titre>titre 3</titre>
    <auteur>auteur 3</auteur>
    <editeur>editeur 3</editeur>
  </livre>
</bibliotheque>
```

La classe OutputFormat possède une méthode createPrettyPrint() qui renvoie un objet de type OutputFormat contenant des paramètres par défaut.

Il est possible d'obtenir une chaine de caractères à partir de tout ou partie d'un document

Exemple :

```
import org.dom4j.*;
import org.dom4j.io.*;
public class Testdom4j_7 {
  public static void main(String args[]) {
    Document document = DocumentHelper.createDocument();
    Element root = document.addElement( "bibliotheque" );
    Element livre = null;
    String texte = "";
    try {
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 1");
      livre.addElement("auteur").addText("auteur 1");
      livre.addElement("editeur").addText("editeur 1");
      texte = document.asXML();
      System.out.println(texte);
    } catch (Exception e){
      e.printStackTrace();
    }
  }
```

```
}
```

Resultat :

```
C:\test_dom4j>java -cp .;c:\j2sdk1.4.0_01/jre/lib/ext/dom4j
-full.jar Testdom4j_7
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque><livre><titre>titre 1</titre><auteur>auteur 1</auteur><editeur>edi
teur 1</editeur></livre></bibliotheque>
```

Cette section est en cours d'écriture

30.13. Jaxen

jaxen

Jaxen est un moteur Xpath qui permet de retrouver des informations grace à Xpath dans un document XML de type dom4j ou Jdom.

C’est un projet open source qui a été intégré dans dom4j pour permettre le support de Xpath dans ce framework.

31. JNDI (Java Naming and Directory Interface)

JNDI est l'acronyme de Java Naming and Directory Interface. Cet API fournit une interface unique pour utiliser différents services de nommages ou d'annuaires:

- LDAP (Lightweigth Directory Access Protocol)
- DNS (Domain Naming Service)
- NIS (Network Information Service) de SUN
- service de nommage CORBA
- service de nommage RMI
- etc ...

Un service de nommage permet d'associer un nom unique à un objet et facilter ainsi l'obtension de cet objet.

Un annuaire est un service de nommage qui possède en plus une représentation hiérarchique des objets qu'il contient et un mécanisme de recherche.

Pour pouvoir utiliser autant de services différents possédant des protocoles d'accès différent, JNDI utilise des pilotes SPI (Service Provider). Trois de ces pilotes sont fournis en standard :

- LDAP (Lightweigth Directory Access Protocol)
- service de nommage CORBA (COS)
- service de nommage RMI

JNDI est intégré au JDK à partir de sa version 1.3.

JNDI est composé de 6 packages :

Packages	Rôle
javax.naming	Classes et interfaces pour utiliser un service nommage
javax.naming.directory	Classes et interfaces pour utiliser un service d'annuaire
javax.naming.event	Classes et interfaces pour l'émission d'événement lors d'un accès à un service
javax.naming.ldap	Classes et interfaces dédiées pour l'utilisation de LDAP v3
javax.naming.spi	Classes et interfaces dédiées aux Service Provider

Pour plus d'informations sur JNDI : http://java.sun.com/products/jndi

Ce chapitre est en cours d'écriture

31.1. Les concepts de base

31.1.1. La définition d'un annuaire

Un annuaire est un outils qui permet de stocker et de consulter des informations selon un protocole particulier. Un annuaire est plus particulière dédié à la recherche et la lecture d'informations.

31.1.2. Le protocole LDAP

Le protocole LDAP (Ligthweight Directory Access Protocol) a été développé pour être un standard de service d'annuaire.

Un annuaire LDAP peut être utilisé pour :

- l'authentification et le contrôle d'accès des utlisateurs aux applications ou aux ressources.
- l'obtension d'informations sur un élément contenu dans l'annuaire
- l'obtension d'un objet stocké dans l'annuaire

Un annuaire LDAP stocke les informations sous la forme d'une arborescence hiérarchique. Le modèle de cette représentation est stocké dans un schéma. Ceci permet de personnaliser l'arborescence. Le premier élément de l'arborescence est nommé racine.

Chaque élément de l'arborescence est défini par un attribut et une valeur. L'attribut est défini dans le schéma et la valeur est libre.

LDAP défini en standard plusieurs attributs :

Attribut	Rôle

Pour accéder à un élément particulier, il faut préciser chaque paire attribut/valeur de chaque élément appartenant à l'arborescence pour accéder à l'élément. Cet ensemble de paire attribut/valeur séparée par une virgule est nommé Dn (Distinguished Name)

31.2. Présentation de JNDI

JNDI est un composant important de J2EE car plusieurs technologies comme les EJB utilise JNDI. JNDI est utilisé pour stocker et obtenir un objet Java. D'autres technologies comme JDBC ou JMS peuvent utiliser JNDI pour les mêmes raisons.

31.3. Utilisation de JNDI avec un serveur LDAP

La première chose à faire est de se connecter au serveur. Il faut utiliser un objet InitialDirContext pour obtenir un objet qui implémente l'interface DirContext. Le constructeur de l'objet InitialDirContext attend un objet de type Hashtable qui contient les paramètres nécessaires à la connexion.

32. JMS (Java Messaging Service)

JMS, aconyme de Java Messaging Service, est une API fournie par Sun pour permettre un dialogue standard entre des applications ou des composants via des brokers de messages ou MOM (Middleware Oriented Messages). Elle permet donc d'utiliser des services de messaging dans des applications java comme le fait l'API JDBC pour les bases de données

Des informations utiles sur JMS peuvent être trouvées à l'URL : http://java.sun.com/products/jms/index.htm

32.1. Présentation de JMS

JMS a été intégré à la plateforme J2EE à partir de la version 1.3 mais il n'existe pas d'implémentation officielle de cette API avant la version 1.3 du J2EE. JMS est utilisable avec les versions antérieures mais elle oblige a utiliser un outils externe qui implémente l'API.

Il existe un certain nombre d'outils qui implémentent JMS dont la majorité sont des produits commerciaux.

Dans la version 1.3 du J2EE, JMS peut être utilisé dans un composant web ou un EJB, un type d'EJB particulier a été ajouté pour traiter les messages et des échanges JMS peuvent être intégrés dans une transaction gérée avec JTA (Java Transaction API).

JMS définit plusieurs entités :

- Un provider JMS : outil qui impémente l'API JMS pour échanger les messages : ce sont les brokers de messages
- Un client JMS : composant écrit en java qui utilise JMS pour émettre et/ou recevoir des messages.
- Un message : données échangées entre les composants

Les messages sont asynchrones mais JMS définit deux modes pour consommer un message :

- Mode synchrone : ce mode nécessite l'appel de la méthode receive() ou d'une de ces surcharges. Dans ce cas, l'application est arrêtée jusqu'à l'arrivée du message. Une version surchargée de cette méthode permet de rendre la main après un certain timeout.
- Mode asynchrone : il faut définir un listener qui va lancer un thread qui va attendre les messages et éxecuter une méthode lors de leur arrivée.

32.2. Les services de messages

Les brokers de messages ou MOM (Middleware Oriented Message) permettent d'assurer l'échange de messages entre deux composants nommés clients. Ces échanges peuvent se faire dans un contexte interne (pour l'EAI) ou un contexte externe (pour le B2B).

Les deux clients n'échangent pas directement des messages : un client envoi un message et le client destinataire doit demander la reception du message. Le transfert du message et sa persistence sont assurés par le broker.

Les échanges de message sont :

- asynchrones :
- fiables : les messages ne sont délivrés qu'un et une seule fois

Les MOM représentent le seul moyen d'effectuer un échange de messages asynchrones. Ils peuvent aussi être très pratique pour l'échange synchrone de messages plutôt que d'utiliser d'autres méchanismes plus compliqués à mettre en œuvre (sockets, RMI, CORBA …).

Les brokers de messages peuvent fonctionner selon deux modes :

- le mode point à point (queue)
- le mode publication/abonnement (publish/souscribe)

Le mode point à point (point to point) repose sur le concept de files d'attente (queues). Le message est stocké dans une file d'attente puis il est lu dans cette file ou dans une autre. Le transfert du message d'une file à l'autre est réalisé par le broker de message.

Chaque message est envoyé dans une seule file d'attente. Il y reste jusqu'à ce qu'il soit consommé par un client et un seul. Le client peut le consommer ultérieurement : la persistance est assurée par le broker de message.

Le mode publication/abonnement repose sur le concept de sujets (Topics). Plusieurs clients peuvent envoyer des messages dans ce topic. Le broker de message assure l'acheminement de ce message à chaque client qui se sera préalablement abonné à ce topic. Le message possède donc potenciellement plusieurs destinataires. L'émetteur du message ne connaît pas les destinataires qui se sont abonnés.

Les principaux brokers de messages commerciaux sont :

Produit	Société	URL
Sonic MQ	Progress software	http://www.progress.com/sonicmq/index.htm
VisiMessage	Borland	http://www.borland.com/appserver
Swift MQ		http://www.swiftmq.com
MessageQ	BEA	http://www.bea.com/products/messageq/datasheet.shtml
MQ Series	IBM	http://www−4.ibm.com/software/ts/mqseries
Rendez vous	Tibco	http://www.tibco.com/products/rv/index.html

Il existe quelques brokers de messages open source :

Outils	URL
OpenJMS	
Joram	

32.3. Le package javax.jms

Ce package et ses sous packages contiennent plusieurs interfaces qui définissent l'API.

- Connection
- Session

- Message
- MessageProducer
- MessageListener

32.3.1. La factory de connexion

Un objet factory est un objet qui permet de retourner un objet pour ce connecter au broker de messages.

Il faut fournir un certain nombre de paramètres à l'objet factory.

Il existe deux types de factory, QueueConnectionFactory et TopicConnectionFactory selon le type d'échanges que l'on fait. Ce sont des interfaces que le broker de message doit implémenter pour fournir des objets.

Pour obtenir un objet de ce type, il faut soit instancier directement un tel objet soit faire appel à JNDI pour l'obtenir.

32.3.2. L'interface Connection

Cette interface définie des méthodes pour la connection au broker de messages.

Cette connection doit être établie en fonction du mode utilisé :

- l'interface QueueConnection pour le mode point à point
- l'interface TopicConnection pour le mode publication/abonnement

Pour obtenir l'un ou l'autre, il faut utiliser un objet factory correspondant de type QueueConnectionFactory ou TopicConnectionFactory avec la méthode correspondante : createQueueConnection() ou createTopicConnection().

La classe qui implémente cette interface se charge du dialogue avec le broker de message.

La méthode start() permet de démarrer la connexion.

Exemple :
`connection.start();`

La méthode stop() permet de suspendre temporairement la connexion.

La méthode close() permet de fermer la connexion.

32.3.3. L'interface Session

Elle représente un contexte transactionnel de réception et d'émission pour une connection donnée.

C'est d'ailleurs à partir d'un objet de type Connection que l'on créé une ou plusieurs sessions.

La session est mono thread : si l'application utilise plusieurs threads qui échangent des messages, il faut définir une session pour chaque thread.

C'est à partir d'un objet session que l'on créé des messages et des objets pour les envoyer et les recevoir.

Comme pour la connexion, la création d'un objet de type Session dépend du mode de fonctionnement. L'interface Session possède deux interfaces filles :

- l'interface QueueSession pour le mode point à point

- l'interface TopicSession pour le mode publication/abonnement

Pour obtenir l'un ou l'autre, il faut utiliser un objet Connection correspondant de type QueueConnection ou TopicConnection avec la méthode correspondante : createQueueSession() ou createTopicSession().

Ces deux méthodes demandent deux paramètres : un boolean qui indique si la session gère une transaction, et une constante qui précise le mode d'accusé de reception des messages.

Il existe trois modes d'accusés de reception (trois constantes sont définies dans l'interface Session) :

- AUTO_ACKNOWLEDGE : l'accusé de reception est automatique
- CLIENT_ACKNOWLEDGE : c'est le client qui envoi l'accusé grace à l'appel de la méthode acknowledge() du message
- DUPS_OK_ACKNOWLEDGE : ce mode permet de dupliquer un message

L'interface Session définie plusieurs méthodes dont les principales:

Méthode	Rôle
void close()	fermer la session
void commit()	valider la transaction
XXX createXXX()	permet de créer un Message dont le type est XXX
void rollback()	Invalide la transaction

32.3.4. Les messages

Ils doivent obligatoirement implémenter l'interface Message ou l'une de ces sous classes.

Les messages sont composés de trois parties :

- l'en tête (header)
- les propriétés (properties)
- le corps du message (body)

32.3.4.1 L'en tête

Cette partie du message contient un certain nombre de champs prédéfinis qui contiennent des données pour identifier et acheminer le message.

La plupart de ces données sont renseignées lors de l'appel à la méthode send() ou publish().

Les champs les plus importants sont :

Nom	Rôle
JMSMessageID	identifiant unique du message
JMSDestination	file d'attente ou topic destinataire du message
JMSCorrelationID	utilisé pour synchroniser de façon applicative deux messages de la forme requête/réponse. Dans ce cas, dans le message réponse, ce champ contient le messageID du message requête

32.3.4.2. Les propriétés

Ce sont des champs supplémentaires : certains sont définis par JMS mais il est possible d'ajouter ces propres champs.

Cette partie du message est optionnelle.

Elles permettent de définir des données qui seront utilisées pour fournir des données supplémentaires ou pour filtrer le message.

32.3.4.3. Le corps du message

Il contient les données du message : ils sont formattés selon le type du message.

Cette partie du message est optionnelle.

Les messages peuvent être de plusieurs types, définis dans les interfaces suivantes :

type	Interface	Role
bytes	BytesMessage	échange d'octets
texte	TextMessage	échange de données texte (XML par exemple)
object	ObjectMessage	échange d'objet java qui doivent être sérialisable
Map	MapMessage	échange de données sous la forme clé/valeur. La clé doit être une chaine de caractères et la valeur de type primitive
Stream	StreamMessage	échange de données en provenance d'un flux

Il est possible de définir son propre type qui doit obligatoirement implémenter l'interface Message.

C'est un objet de type Session qui contient les méthodes nécessaires à la création d'un message selon son type.

Lors de la réception d'un message, celui ci est toujours de type Message : il faut effectuer un transtypage en fonction de son type en utilisant l'opérateur instanceof. A ce moment, il faut utiliser le getter correspondant pour obtenir les données.

Exemple :

```
Message message = ...

if (message instanceof TextMessage) {
   TextMessage textMessage = (TextMessage) message;
   System.out.println("message: " + textMessage.getText());
}
```

32.3.5. L'envoi de Message

L'interface MessageProducer est la super interface des interfaces qui définissent des méthodes pour l'envoi de messages.

Il exite deux interfaces filles selon le mode de fonctionnement pour envoyer un message : QueueSender et TopicPublisher.

Ces objets sont créés à partir d'un objet représentant la session :

- la méthode createSender() pour obtenir un objet de type QueueSender
- la méthode createPublisher() pour obtenir un objet de type TopicPublisher

Ces objets peuvent être liés à une entité physique par exemple une file d'attente particulière pour un objet de type QueueSender. Si ce n'est pas le cas, cette entité devra être précisée lors de l'envoie du message en utilisant une version surchargée de la méthode chargée de l'emission du message.

32.3.6. La réception de messages

L'interface MessageConsumer est la super interface des interfaces qui définissent des méthodes pour la réception de messages.

Il existe des interfaces selon le mode fonctionnement pour recevoir un message QueueReceiver et TopicSubscriber.

La réception d'un message peut se faire avec deux modes :

- synchrone : dans ce cas, l'attente d'un message bloque l'execution du reste de code
- asynchrone : dans ce cas, un thread est lancé qui attend le message et appelle une méthode (callback) à son arrivée. L'execution de l'application n'est pas bloquée.

L'interface MessageConsumer définie plusieurs méthodes sont les principales sont :

Méthode	Rôle
close()	fermer l'objet qui reçoit les messages pour le rendre inactif
Message receive()	attend et retourne le message à son arrivée
Message receice(long)	attend durant le nombre de milliseconde précisé en paramètre et renvoie le message si il arrive durant ce laps de temps
Message receiveNoWait()	renvoie un message sans attendre si il y en a un de présent
setMessageListener(MessageListener)	associe un Listener pour traiter les messages de façon asynchrone

Pour obtenir un objet qui implémente l'interface QueueReceiver, il faut utiliser la méthode createReceiver() d'un objet de type QueueSession.

Pour obtenir un objet qui implémente l'interface TopicSubscriber, il faut utiliser la méthode createSubscriber() d'un objet de type TopicSession.

32.4. L'utilisation du mode point à point (queue)

32.4.1. La création d'une factory de connexion : QueueConnectionFactory

Un objet factory est un objet qui permet de retourner un objet pour ce connecter au broker de messages.

Pour obtenir un objet de ce type, il faut soit instancier directement un tel objet soit faire appel à JNDI pour l'obtenir.

Exemple avec MQSeries :

```
String qManager = ...
String hostName = ...
String channel = ...

MQQueueConnectionFactory factory = new MQQueueConnectionFactory();
factory.setQueueManager(qManager);
```

```
factory.setHostName(hostName);
factory.setChannel(channel);
factory.setTransportType(JMSC.MQJMS_TP_CLIENT_MQ_TCPIP);
```

32.4.2. L'interface QueueConnection

Cette interface hérite de l'interface Connection.

Pour obtenir un objet qui implémente cette interface, il faut utiliser un objet factory correspondant de type QueueConnectionFactory avec la méthode correspondante : createQueueConnection().

Exemple :

```
QueueConnection connection = factory.createQueueConnection();
connection.start();
```

L'interface QueueConnection définie plusieurs méthodes dont la principale est :

Méthode	Rôle
QueueSession createQueueSession(boolean, int)	renvoie un objet qui définie la session. Le boolean précise si la session gère une transaction. L'entier précise le mode d'accusé de reception.

32.4.3. La session : l'interface QueueSession

Elle hérite de l'interface Session.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createQueueSession() d'un objet connexion de type QueueConnection.

Exemple :

```
QueueSession session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
```

L'interface QueueSession définie plusieurs méthodes sont les principales sont :

Méthode	Rôle
QueueReceiver createQueueReceiver(Queue)	renvoie un objet qui définie une file d'attente de réception
QueueSender createQueueSender(Queue)	renvoie un objet qui définie une file d'attente d'émission

32.4.4. L'interface Queue

Un objet qui implémente cette interface encapsule une file d'attente particulière.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createQueue() d'un objet de type QueueSession.

Exemple avec MQseries :

```
Queue fileEnvoi =
   session.createQueue("queue:///file.out"?expiry=0&persistence=1&targetClient=1");
```

32.4.5. La création d'un message

Pour créer un message, il faut utiliser une méthode createXXXMessage() d'un objet QueueSession ou XXX représente le type du message.

Exemple :

```
String message = «bonjour»;
TextMessage textMessage = session.createTextMessage();
textMessage.setText(message);
```

32.4.6. L'envoi de messages : l'interface QueueSender

Cette interface hérite de l'interface MessageProducer.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createQueueSender() d'un objet de type QueueSession.

Exemple :

```
QueueSender queueSender = session.createSender(fileEnvoi);
```

Il est possible de fournir un objet de type Queue qui représente la file d'attente : dans ce cas, l'objet QueueSender est lié à cette file d'attente. Si l'on ne précise pas de file d'attente (null fourni en paramètre), dans ce cas, il faudra obligatoirement utilisée un version surchargée de la méthode send() lors de l'envoi pour préciser la file d'attente.

Avec un objet de type QueueSender, la méthode send() permet l'envoie d'un message dans la file d'attente. Cette méthode possède plusieurs surcharges :

Méthode	Rôle
void send(Message)	Envoie le message dans la file d'attente définie dans l'objet de type QueueSender
void send(Queue, Message)	Envoie le message dans la file d'attente fournie en paramètre

Exemple :

```
queueSender.send(textMessage);
```

32.4.7. La réception de messages : l'interface QueueReceiver

Cette interface hérite de l'interface MessageConsummer.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createQueueReceiver() à partir d'un objet de type QueueSession.

Exemple :

```
QueueReceiver queueReceiver = session.createReceiver(fileReception);
```

Il est possible de fournir un objet de type Queue qui représente la file d'attente : dans ce cas, l'objet QueueSender est lié à cette file d'attente. Si l'on ne précise pas de file d'attente (null fourni en paramètre), dans ce cas, il faudra

obligatoirement utilisée un version surchargé de la méthode receive() lors de l'envoi pour préciser la file d'attente.

Cette interface ne définie qu'une seule méthode supplémentaire :

Méthode	Rôle
Queue getQueue()	renvoie la file d'attente associée à l'objet

La réception de messages peut ce faire dans le mode synchrone ou asynchrone.

32.4.7.1. La réception dans le mode synchrone

Dans ce mode, le programme est interrompu jusqu'à l'arrivée d'un nouveau message. Il faut utiliser la méthode receive() héritée de l'interface MessageConsumer. Il existe plusieurs méthodes et surcharges de ces méthodes qui permettent de faire face à toute les utilisations :

- receiveNoWait() : renvoi un message présent sans attendre
- receive(long) : renvoi un message qui arrive durant le temps fourni en paramètre
- receive() : renvoi le message dès qu'il arrive

Exemple :

```
Message message = null;
message = queueReceiver.receive(10000);
```

32.4.7.2. La réception dans le mode asynchrone

Dans ce mode, le programme n'est pas interrompu mais un objet écouteur va être enregistré auprès de l'objet de type QueueReceiver. Cet objet qui implémente l'interface MessageListener va être utilisé comme gestionnaire d'évenements lors de l'arrivé d'un nouveau message.

L'interface MessageListener ne définie qu'une seule méthode qui reçoit en paramètre le message : onMessage(). C'est cette méthode qui sera appelée lors de la reception d'un message.

32.4.7.3. La sélection de messages

Une version surchargée de la méthode createReceiver() d'un objet de type QueueSession permet de préciser dans ces paramètres une chaîne de caractères qui va servir de filtre sur les messages à recevoir.

Dans ce cas, le filtre est effectué par le broker de message plutôt que par le programme.

Cette chaîne de caractères contient une expression qui doit avoir une syntaxe proche d'une condition SQL. Les critères de la sélection doivent porter sur des champs inclus dans l'en tête ou dans les propriétés du message. Il n'est pas possible d'utiliser des données du corps du message pour effectuer le filtre.

Exemple : envoi d'un message requête et attente de sa réponse. Dans ce cas, le champ JMSCorrelationID du message réponse contient le JMSMessageID du message requete

```
String messageEnvoie = «bonjour»;
TextMessage textMessage = session.createTextMessage();
textMessage.setText(messageEnvoie);
queueSender.send(textMessage);
int correlId = textMessage.getJMSMessageID();
QueueReceiver queueReceiver = session.createReceiver(
fileEnvoie, "JMSCorrelationID = '" + correlId +"'");
```

```
Message message = null;
message = queueReceiver.receive(10000);
```

32.5. L'utilisation du mode publication/abonnement (publish/souscribe)

32.5.1. La création d'une factory de connexion : TopicConnectionFactory

Un objet factory est un objet qui permet de retourner un objet pour ce connecter au broker de messages.

Pour obtenir un objet de ce type, il faut soit instancier directement un tel objet soit faire appel à JNDI pour l'obtenir.

32.5.2. L'interface TopicConnection

Cette interface hérite de l'interface Connection.

Pour obtenir un objet qui implémente cette interface, il faut utiliser un objet factory correspondant de type TopicConnectionFactory avec la méthode correspondante : createTopicConnection().

Exemple :

```
TopicConnection connection = factory.createTopicConnection();
connection.start();
```

L'interface TopicConnection définie plusieurs méthodes dont la principale est :

Méthode	Rôle
TopicSession createTopicSession(boolean, int)	renvoie un objet qui définie la session. Le boolean précise si la session gère une transaction. L'entier précise le mode d'accusé de reception.

32.5.3. La session : l'interface TopicSession

Elle hérite de l'interface Session.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createTopicSession() d'un objet connexion de type TopicConnection.

Exemple :

```
TopicSession session = connection.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
```

L'interface TopicSession définie plusieurs méthodes dont les principales sont :

Méthode	Rôle
TopicSubscriber createSubscriber(Topic)	renvoie un objet qui permet l'envoie de messages dans un topic
TopicPublisher createPublisher(Topic)	renvoie un objet qui permet la reception de messages dans un topic

Topic createTopic(String)	création d'un topic correspondant à la désignation fournie en paramètre

32.5.4. L'interface Topic

Un objet qui implémente cette interface encapsule un sujet.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createTopic() d'un objet de type TopicSession.

32.5.5. La création d'un message

Pour créer un message, il faut utiliser une méthode createXXXMessage() d'un objet TopicSession ou XXX représente le type du message.

Exemple :

```
String message = «bonjour»;
TextMessage textMessage = session.createTextMessage();
textMessage.setText(message);
```

32.5.6. L'émission de messages : l'interface TopicPublisher

Cette interface hérite de l'interface MessageProducer.

Avec un objet de type TopicPublisher, la méthode publish() permet l'envoie du message. Cette méthode possède plusieurs surcharges :

Méthode	Rôle
void publish(Message)	envoie le message dans le topic défini dans l'objet de type TopicPublisher
void publish(Topic, Message)	envoie le message dans le topic fourni en paramètre

32.5.7. La réception de messages : l'interface TopicSubscriber

Cette interface hérite de l'interface MessageProducer.

Pour obtenir un objet qui implémente de cette interface, il faut utiliser la méthode createSubscriber() à partir d'un objet de type TopicSession.

Exemple :

```
TopicSubscriber topicSubscriber = session.createSubscriber(topic);
```

Il est possible de fournir un objet de type Topic qui représente le topic : dans ce cas, l'objet Topicsubscriber est lié à ce topic. Si l'on ne précise pas de topic (null fourni en paramètre), dans ce cas, il faudra obligatoirement utilisée un version surchargé de la méthode receive() lors de l'envoi pour préciser le topic.

Cette interface ne définit qu'une seule méthode supplémentaire :

Méthode	Rôle

Topic getTopic()	renvoie le topic associée à l'objet

32.6. Les exceptions de JMS

Plusieurs exceptions sont définies par l'API JMS. La classe mère de toute ces exceptions est la classe JMSException.

Les exceptions définies sont : IllegalStateException, InvalidClientIDException, InvalidDestinationException, InvalidSelectorException, JMSSecurityException, MessageEOFException, MessageFormatException, MessageNotReadableException, MessageNotWriteableException, ResourceAllocationException, TransactionInProgressException,TransactionRolledBackException

La méthode getErrorCode() permet d'obtenir le code erreur spécifique du produit sous forme de chaîne de caractères.

Un exemple avec OpenJMS

33. JavaMail

Le courrier électronique repose sur le concept du client/serveur. Ainsi, l'utilisation d'e mail requiert deux composants :

- un client de mail (Mail User Agent : MUA) tel que Outlook, Messenger, Eudora ...
- un serveur de mail (Mail Transport Agent : MTA) tel que SendMail

Les clients de mail s'appuie sur un serveur de mail pour obtenir et envoyer des messages. Les échanges entre client et serveur sont normalisés par des protocoles particuliers.

JavaMail est une API qui permet d'utiliser le courrier électronique (e−mail) dans une application écrite en java (application cliente, applet, servlet, EJB ...). Son but est d'être facile à utiliser, de fournir une souplesse qui permette de la faire évoluer et de rester le plus indépendant possible des protocoles utilisés.

JavaMail est une extension au JDK qui n'est donc pas fournie avec J2SE. Pour l'utiliser, il est possible de la télécharger sur le site de SUN : http://java.sun.com/products/javamail. Elle est intégré au J2EE.

Les classes et interfaces sont regroupées dans quatre packages : javax.mail, javax.mail.event, javax.mail.internet, javax.mail.search.

Il existe deux versions de cette API :

- 1.1.3 : version fournie avec J2EE 1.2
- 1.2 : version courante

Les deux versions fonctionnent avec un JDK dont la version est au moins 1.1.6.

Cette API permet une abstraction assez forte de tout système de mail, ce qui lui permet d'ajouter des protocoles non gérés en standard. Pour gérer ces différents protocoles, il faut utiliser une implémentation particulière pour chacun d'eux, fournis par des fournisseurs tiers. En standard, JavaMail 1.2 fournie une implémentation pour les protocoles SMTP, POP3 et IMAP4. JavaMail 1.1.3 ne fournie une implémentation que pour les protocoles SMTP et IMAP : l'implémentation pour le protocole POP3 doit être téléchargé séparément.

33.1. Téléchargement et installation

Pour le J2SE, il est nécessaire de télécharger les fichiers utiles et de les installer.

Pour les deux versions de l'API, il faut télécharger la version correspondante, dezipper le fichier dans un répertoire et ajouter le fichier mail.jar dans le CLASSPATH.

Ensuite il faut aussi installer le framework JAF (Java Activation Framework) : télécharger le fichier, dezipper et ajouter le fichier activation.jar dans le CLASSPATH

Pour pouvoir utiliser le protocole POP3 avec JavaMail 1.1.3, il faut télécharger en plus l'implémentation de ce protocole et inclure le fichier POP3.jar dans le CLASSPATH.

Pour le J2EE 1.2.1, l'API version 1.1.3 est intégrée à la plate−forme. Elle ne contient donc pas l'implémentation pour le protocole POP3. Il faut la télécharger et l'installer en plus comme avec le J2SE.

Pour le J2EE 1.3, il n'y a rien de particulier à faire puisque l'API version 1.2 est intégrée à la plate−forme.

33.2. Les principaux protocoles

33.2.1. SMTP

SMTP est l'acronyme de Simple Mail Transport Protocol. Ce protocole défini par la recommandation RFC 821 permet l'envoie de mail vers un serveur de mail qui supporte ce protocole.

33.2.2. POP

POP est l'acronyme de Post Office Protocol. Ce protocole défini par la recommandation RFC 1939 permet la réception de mail à partir d'un serveur de mail qui supporte ce protocole. La version courante de ce protocole est 3. C'est un protocole très populaire sur Internet. Il définit une boite au lettre unique pour chaque utilisateur. Une fois que le message est reçu par le client, il est effacé du serveur.

33.2.3. IMAP

IMAP est l'acronyme de Internet Message Acces Procol. Ce protocole défini par la recommandation RFC 2060 permet aussi la réception de mail à partir d'un serveur de mail qui supporte ce protocole. La version courante de ce protocole est 4. Ce protocole est plus complexe car il apporte des fonctionnalités supplémentaires : plusieurs répertoires par utilisateur, partage de répertoire entre plusieurs utilisateurs, maintient des messages sur le serveur , etc ...

33.2.4. NNTP

NNTP est l'acronyme de Network News Transport Protocol. Ce protocole est utilisé par les forums de discussion (news).

33.3. Les principales classes et interface de l'API JavaMail

JavaMail propose des classes et interfaces qui encapsulent ou définissent les objets liés à l'utilisation des mails et les protocoles utilisés pour les échanger.

33.3.1. La classe Session

La classe Session encapsule pour un client donné sa connexion avec le serveur de mail. Cette classe encapsule les données liées à la connexion (options de configuration et données d'authentification). C'est à partir de cet objet que toutes les actions concernant les mails sont réalisées.

Les paramètres nécessaires sont fournis dans un objet de type Properties. Un objet de ce type est utilisé pour contenir les variables d'environnements : placer certaines informations dans cet objet permet de partager des données.

Une session peut être unique ou partagée par plusieurs entités.

Exemple :

```
// creation d'une session unique
Session session = Session.getInstance(props,authenticator);
// creation d'une session partagee
Session defaultSession = Session.getDefaultInstance(props,authenticator);
```

Pour obtenir une session, deux paramètres sont attendus :

- un objet Properties qui contient les paramètres d'initialisation. Un tel objet est obligatoire
- un objet Authenticator optionnel qui permet d'authentifier l'utilisateur auprès du serveur de mail

La méthode setDebug() qui attend en paramètre un booleen est très pratique pour debugger car avec le paramètre true, elle affiche des informations lors de l'utilisation de la session notamment le détail des commandes envoyées au serveur de mail.

33.3.2. Les classes Address, InternetAddress et NewsAddress

La classe Address est une classe abstraite dont héritent toutes les classes qui encapsulent une adresse dans un message.

Deux classes filles sont actuellement définies :

- InternetAddress
- NewsAddress

Le classe InternetAddress encapsule une adresse email respectant le format de la RFC 822. Elle contient deux champs : address qui contient l'adresse e mail et personal qui contient le nom de la personne. La classe possède des constructeurs, des getters et des setters pour utiliser ces attributs.

Le plus simple pour créer un objet InternetAddress est d'appeler le constructeur en lui passant en paramètre une chaîne de caractère contenant l'adresse e−mail.

Exemple :

```
InternetAddress vInternetAddresses = new InternetAddress;
vInternetAddresses = new InternetAddress("moi@chez−moi.fr");
```

Un second constructeur permet de préciser l'adresse e−mail et un nom en clair.

La méthode getLocalAddress(Session) permet de déterminer si possible l'objet InternetAddress encapsulant l'adresse e mail de l'utilisateur courant, sinon elle renvoie null.

La méthode parse(String) permet de créer un tableau d'objet InternetAddress à partir d'une chaîne contenant les adresses e mail séparées par des virgules.

Un objet InternetAddress est nécessaire pour chaque émetteur et destinataire du mail. L'API ne vérifie pas l'existence des adresses fournies. C'est le serveur de mail qui vérifiera les destinataires et éventuellement les émetteurs selon son paramétrage.

La classe NewsAddress encapsule une adresse news (forum de discussion) respectant le format RFC1036. Elle contient deux champs : host qui contient le nom du serveur et newsgroup qui le nom du forum

La classe possède des constructeurs, des getters et des setters pour utiliser ces attributs.

33.3.3. L'interface Part

Cette interface définie un certain nombre d'attributs commun à la plupart des systèmes de mail et un contenu.

Le contenu peut être renvoyé sous trois formes : DataHandler, InputStream et Object.

Cette interface définie plusieurs méthodes principalement des getters et des setters donc les principaux sont :

Méthode	Rôle
int getSize()	Renvoie la taille du contenu sinon −1 si elle ne peut être déterminée
int getLineCount()	Renvoie le nombre de ligne du contenu sinon −1 s'il ne peut être déterminé
String getContentType()	Renvoie le type du contenu sinon null
String getDescription()	Renvoie la description
void setDescription(String)	Mettre à jour la description
InputStream getInputStream()	Renvoie le contenu sous le forme d'un flux
DataHandler getDataHandler()	Renvoie le contenu sous la forme d'un objet DataHandler
Object getContent()	Renvoie le contenu sous la forme d'un objet. Un cast est nécessaire selon le type du contenu.
void setText(String)	Mettre à jour le contenu sous forme d'une chaîne de caractères fournie en paramètre

33.3.4. La classe Message

La classe abstraite Message encapsule un Message. Le message est composé de deux parties :

- une en−tête qui contient des attributs
- un corps qui contient les données à envoyer

Pour la plupart de ces données, la classe Message implémente l'interface Part qui encapsule les attributs nécessaires à la distribution du message (auteur, destinataire, sujet …) et le corps du message.

Le contenu du message est stocké sous forme d'octet. Pour accéder à son contenu, il faut utiliser un objet du JavaBean Activation Framework (JAF) : DataHandler. Ceci permet une séparation des données nécessaires à la transmission et du contenu du message qui peut ainsi prendre n'importe quel format. La classe Message ne connaît pas directement le type du contenu du corps du message.

JavaMail fourni en standard une classe fille nommée MimeMessage qui implémente la recommandation RFC 822 pour les messages possédant un type Mime.

Il y a deux façons d'obtenir un objet de type Message : instancier une classe fille pour créer un nouveau message ou utiliser un objet de type Folder pour obtenir un message existant.

La classe Message défini deux constructeurs en plus du constructeur par défaut :

Constructeur	Rôle
Message(session)	Créer un nouveau message
Message(Folder, int)	Créer un message à partir d'un message existant

La classe MimeMessage est la seule classe fille qui hérite de la classe Message. Elle dispose de plusieurs constructeurs.

Exemple :

```
MimeMessage message = new MimeMessage(session);
```

Elle possèdent de nombreuses méthodes pour initialiser les données du message :

Méthode	Rôle
void addFrom(Address[])	Ajouter des émetteurs au message
void addRecipient(RecepientType, Address[])	Ajouter des destinataires à un type (direct, en copie ou en copie cachée)
Flags getFlags()	Renvoie les états du message
Adress[] getFrom()	Renvoie les émetteurs
int getLineCount()	Renvoie le nombre ligne du message
Address[] getRecipients(RecepientType)	Renvoie les destinataires du type fourni en paramètre
Address getReplyTo()	Renvoie les email pour la réponse
int getSize()	Renvoie la taille du message
String getSubject()	Renvoie le sujet
Message reply(boolean)	Créer un message pour la réponse : le booleen indique si la réponse ne doit être faite qu'à l'émetteur
void set content(Object, String)	Mettre à jour le contenu du message en précisant son type mime
void seFrom(Address)	Mettre à jour l'émetteur
void setRecipients(RecepientsType, Address[])	Mettre à jour les destinataires d'un type
void setSendDate(Date)	Mettre à jour la date d'envoi
void setText(String)	Mettre à jour le contenu du message avec le type mime « text/plain »
void setReply(Address)	Mettre à jour le destinataire de la réponse
void writeTo(OutputStream)	Envoie le message au format RFC 822 dans un flux. Très pratique pour visualiser le message sur la console en passant en paramètre (System.out)

La méthode addRecipient() permet d'ajouter un destinataire et le type d'envoi.

Le type d'envoi est précisé grâce une constante pour chaque type :

- destinataire direct : Message.RecipientType.TO
- copie conforme : Message.RecipientType.CC
- copie cachée : Message.RecipientType.BCC

La méthode setText() permet de facilement mettre une chaîne de caractères dans le corps du message avec un type MIME « text/plain ». Pour envoyer un message dans un format différent, par exemple HTML, il utilise la méthode

setContent() qui attend en paramètre un objet et un chaîne qui contient le type MIME du message.

Exemple :

```
String texte = "<H1>bonjour</H1><a
    href=\"mailto:moi@moi.fr\">mail</a>";
message.setContent(texte, "text/html");
```

Il est possible de joindre avec le mail des ressources sous forme de pièces jointes (attachments). Pour cela, il faut :

- instancier un objet de type MimeMessage
- renseigner les éléments qui composent l'en−tête : émetteur, destinataire, sujet ...
- Instancier un objet de type MimeMultiPart
- Instancier un objet de type MimeBodyPart et alimenter le contenu de l'élément
- Ajouter cet objet à l'objet MimeMultiPart grâce à la méthode addBodyPart()
- Répéter l'instanciation et l'alimentation pour chaque ressource à ajouter
- utiliser la méthode setContent() du message en passant en paramètre l'objet MimeMultiPart pour associer le message et les pièces jointes au mail

Exemple :

```
Multipart multipart = new MimeMultipart();

// creation partie principale du message
BodyPart messageBodyPart = new MimeBodyPart();
messageBodyPart.setText("Test");
multipart.addBodyPart(messageBodyPart);

// creation et ajout de la piece jointe
messageBodyPart = new MimeBodyPart();
DataSource source = new FileDataSource("image.gif");
messageBodyPart.setDataHandler(new DataHandler(source));
messageBodyPart.setFileName("image.gif");
multipart.addBodyPart(messageBodyPart);

// ajout des éléments au mail
message.setContent(multipart);
```

33.3.5. Les classes Flags et Flag

Cette classe encapsule un ensemble d'états pour un message.

Il existe deux type d'états : les états prédéfinis (System Flag) et les états particuliers définis par l'utilisateur (User Defined Flag)

Un état prédéfini est encapsulé par la classe Internet Flags.Flag. Cette classe définie plusieurs états statiques :

Etat	Rôle
Flags.Flag.ANSWERED	Le message a été demandé : positionné par le client
Flags.Flag.DELETED	Le message est marqué pour la suppression
Flags.Flag.DRAFT	Le message est un brouillon
Flags.Flag.FLAGGED	Le message est marqué dans un état qui n'a pas de définition particulière
Flags.Flag.RECENT	Le message est arrivé recemment. Le client ne peut pas modifier cet état.

Flags.Flag.SEEN	Le message a été visualisé : positionné à l'ouverture du message
Flags.Flag.USER	Le client a la possibilité d'ajouter des états particuliers

Tous ces états ne sont pas obligatoirement supportés par le serveur.

La classe Message possède plusieurs méthodes pour gérer les états d'un message. La méthode getFlags() renvoie un objet Flag qui contient les états du message. Les méthodes setFlag(Flag, boolean) permettent d'ajouter un état du message. La méthode contains(Flag) vérifie si l'état fourni en paramètre est positionné pour le message.

La classe Flags possède plusieurs méthodes pour gérer les états dont les principales sont :

Méthode	Rôle
void add(Flags.Flag)	Permet d'ajouter un état
void add(Flags)	Permet d'ajouter un ensemble d'état
void remove(Flags.Flag)	Permet d'enlever un état
void remove(Flags)	Permet d'enlever un ensemble d'état
boolean contains(Flags.Flag)	Permet de savoir si un état est positionné

33.3.6. La classe Transport

La classe Transport se charge de réaliser l'envoie du message avec le protocole adéquat. C'est une classe abstraite qui contient la méthode static send() pour envoyer un mail.

Il est possible d'obtenir un objet Transport dédié au protocole particulier utilisé par la session en utilisant la méthode getTransport() d'un objet Session. Dans ce cas, il faut :

1. établir la connexion en utilisant la méthode connect() avec le nom du serveur, le nom de l'utilisateur et son mot de passe
2. envoyer le message en utilisant la méthode sendMessage() avec le message et les destinataires. La méthode getAllRecipients() de la classe message permet d'obtenir ceux contenus dans le message.
3. fermer la connexion en utilisant la méthode close()

Il est préférable d'utiliser une instance de Transport tel qu'expliqué ci dessus lorsqu'il y a plusieurs mails à envoyer car on peut maintenir la connexion avec le serveur ouverte pendant les envois.

La méthode static send() ouvre et ferme la connexion à chacun de ces appels.

33.3.7. La classe Store

La classe abstraite store qui représente un système de stockage de messages. Pour obtenir une instance de cette classe, il faut utiliser la méthode getStore() d'un objet de type Session en lui donnant comme paramètre le protocole utilisé.

Pour pouvoir dialoguer avec le serveur de mail, il faut appeler la méthode connect() en lui précisant le nom du serveur, le nom d'utilisateur et le mot de passe de l'utilisateur.

La méthode close() permet de libérer la connexion avec le serveur.

33.3.8. La classe Folder

La classe abstraite Folder représente un répertoire dans lequel les messages sont stockés. Pour obtenir un instance de cette classe, il faut utiliser la méthode getFolder() d'un objet de type Store en lui précisant le nom du répertoire.

Avec le protocole POP3 qui ne gère qu'un seul répertoire, le seul possible est « INBOX ».

Pour pouvoir être utilisé, il faut appeler la méthode open() de la classe Folder en lui précisant le mode d'utilisation : READ_ONLY ou READ_WRITE.

Pour obtenir les messages contenus dans le répertoire, il faut appeler la méthode getMessages(). Cette méthode renvoie un tableau de Message qui peut être null si aucun message n'est renvoyé.

Une fois les opérations terminées, il faut fermer le répertoire en utilisant la méthode close().

33.3.9. Les propriétés d'environnement

JavaMail utilise des propriétés d'environnement pour recevoir certains paramètres de configuration. Ils sont stockés dans un objet de type Properties.

L'objet Properties peut contenir un certains nombre de propriétés qui possèdent des valeurs par défaut :

Propriété	Rôle	Valeur par défaut
mail.store.protocol	Protocole de stockage du message	le premier protocole concerné dans le fichier de configuration
mail.transport.protocol	Protocole de transport par défaut	le premier protocole concerné dans le fichier de configuration
mail.host	Serveur de Mail par défaut	localhost
mail.user	Nom de l'utilisateur pour se connecter au serveur de mail	user.name
mail.protocol.host	Serveur de mail pour un protocole dédié	mail.host
mail.protocol.user	Nom de l'utilisateur pour se connecter au serveur de mail pour un protocole dédié	
mail.from	adresse par défaut de l'expéditeur	user.name@host
mail.debug	mode de debuggage par défaut	

Attention : l'utilisation de JavaMail dans une applet implique de fournir explicitement toutes les valeurs des propriétés utiles car une applet n'a pas la possibilité de définir toutes les valeurs par défaut car l'accès à ces porpriétés est restreint.

L'usage de certains serveurs de mail nécessite l'utilisation d'autres propriétés.

33.3.10 La classe Authenticator

Authenticator est une classe abstraite qui propose des méthodes de base pour permettre d'authentifier un utilisateur. Pour l'utiliser, il faut créer une classe fille qui se chargera de collecter les informations. Plusieurs méthodes appelées selon les besoins sont à redéfinir :

Méthode	Rôle

String getDefaultUserName()	
PasswordAuthentication getPasswordAuthentication()	
int getRequestingPort()	
String getRequestingPort()	
String getRequestingProtocol()	
InetAddress getRequestingSite()	

Par défaut, la méthode getPasswordAuthentication() de la classe Authentication renvoie null. Cette méthode renvoie un objet PasswordAuthentication à partir d'une source de données (boîte de dialogue pour saisie, base de données …).

Une instance d'une classe fille de la classe Authenticator peut être fournie à la session. L'appel à Authenticator sera fait selon les besoins par la session.

33.4. L'envoie d'un e mail par SMTP

Pour envoyer un e mail via SMTP, il faut suivre les principales étapes suivantes :

- Positionner les variables d'environnement nécessaires
- Instancier un objet Session
- Instancier un objet Message
- Mettre à jour les attributs utiles du message
- Appeler la méthode send() de la classe Transport

Exemple :

```
import javax.mail.internet.*;
import javax.mail.*;
import java.util.*;

/**
 * Classe permettant d'envoyer un mail.
 */
public class TestMail {
   private final static String MAILER_VERSION = "Java";
   public static boolean envoyerMailSMTP(String serveur, boolean debug) {
         boolean result = false;
         try {
                  Properties prop = System.getProperties();
                  prop.put("mail.smtp.host", serveur);
                  Session session = Session.getDefaultInstance(prop,null);
                  Message message = new MimeMessage(session);
                  message.setFrom(new InternetAddress("moi@chez-moi.fr"));
                  InternetAddress[] internetAddresses = new InternetAddress[1];
                  internetAddresses[0] = new InternetAddress("moi@chez-moifr");
                  message.setRecipients(Message.RecipientType.TO,internetAddresses);
                  message.setSubject("Test");
                  message.setText("test mail");
                  message.setHeader("X-Mailer", MAILER_VERSION);
                  message.setSentDate(new Date());
                  session.setDebug(debug);
                  Transport.send(message);
                  result = true;
         } catch (AddressException e) {
                  e.printStackTrace();
         } catch (MessagingException e) {
                  e.printStackTrace();
         }
         return result;
```

```
   }

   public static void main(String[] args) {
         TestMail.envoyerMailSMTP("10.10.50.8",true);
   }
}
```

javac −classpath activation.jar;mail.jar;smtp.jar %1.java

java −classpath .;activation.jar;mail.jar;smtp.jar %1

33.5. Récupérer les messages d’un serveur POP3

Cette section est en cours d'écriture

33.6. Les fichiers de configuration

Ces fichiers permettent d’enregistrer des implementations de protocoles supplémentaires et des valeurs par défaut. Il existe 4 fichiers répartis en deux catégories :

- javamail.providers et javamail.default.providers
- javamail.address.map et javamail.default.address.map

JavaMail recherche les informations contenues dans ces fichiers dans l’ordre suivant :

1. $JAVA_HOME/lib

2. META−INF/javamail.xxx dans le fichier jar de l’application

3. MATA−INF/javamail.default.xxx dans le fichier jar de javamail

Il est ainsi possible d’utiliser son propre fichier sans faire de modification dans le fichier jar de JavaMail. Cette utilisation peut se faire sur le poste client ou dans le fichier jar de l’application, ce qui offre une grande souplesse.

33.6.1. Les fichiers javamail.providers et javamail.default.providers

Ce sont deux fichiers au format texte qui contiennent la liste et la configuration des protocoles dont le système dispose d’une implémentation. L’application peut ainsi rechercher la liste des protocoles utilisables.

Chaque procotole est défini en utilisant des attributs avec la forme nom=valeur suivi d’un point virgule. Cinq attributs sont définis (leur nom doit être en minuscule) :

Nom de l’attribut	Rôle	Présence
protocol	nom du protocole	obligatoire

type	type protocole : « store » ou « transport »	obligatoire
class	nom de la classe contenant l'implémentation du protocole	obligatoire
vendor	nom du fournisseur	optionnelle
version	numero de version	optionnelle

Exemple : le contenu du fichier META−INF/javamail.default.providers

```
# JavaMail IMAP provider Sun Microsystems, Inc
protocol=imap; type=store; class=com.sun.mail.imap.IMAPStore; vendor=Sun Microsystems, Inc;
# JavaMail SMTP provider Sun Microsystems, Inc
protocol=smtp; type=transport; class=com.sun.mail.smtp.SMTPTransport; vendor=Sun Microsystems, Inc;
# JavaMail POP3 provider Sun Microsystems, Inc
protocol=pop3; type=store; class=com.sun.mail.pop3.POP3Store; vendor=Sun Microsystems, Inc;
```

33.6.2. Les fichiers javamail.address.map et javamail.default.address.map

Ce sont deux fichiers au format texte qui permettent d'associer un type de transport avec un protocole. Cette association se fait sous la forme nom=valeur suivi d'un point virgule.

Exemple : le contenu du fichier META−INF/javamail.default.address.map

```
rfc822=smtp
```

34. JDO (Java Data Object)

34.1. Présentation

JDO (Java Data Object) est une spécification JCP n° 12 qui propose une technologie pour assurer la persistance d'objets java.

La version 1.0 de cette spécification a été validée au premier trimestre 2002. Elle devrait connaître un grand succès car le mapping entre des données stockées dans un format particulier (bases de données ...) et un objet a toujours été difficile. JDO propose de faciliter cette tache.

Les principaux buts de JDO sont :

- la facilité d'utilisation (gestion automatique du mapping des données)
- la persistance universelle : persistance vers tout type de système de gestion de ressources (bases de données relationnelles, fichiers, ...)
- la transparence vis à vis du système de gestion de ressources utilisé : ce n'est plus le développeur mais JDO qui dialogue avec le système de gestion de ressources
- la standardisation des accès aux données
- la prise en compte des transactions

Le développement avec JDO se déroule en plusieurs étapes :

1. écriture des objets métiers (des beans qui encapsulent les données)
2. écriture des objets qui utilisent les objets métiers pour répondre aux besoins fonctionnelles. Ces objets utilisent l'API JDO.
3. écriture du fichier metadata qui précise le mapping entre les objets et le système de gestion des ressources. Cette partie est très dépendante du système de gestion de ressources utilisé
4. enrichissement des objets métiers
5. configuration du système de gestion des ressources

JDBC et JDO ont les différences suivantes :

JDBC	JDO
orienté SQL	orienté objets
le code doit être ajouté explicitement	code est ajouté automatique
	gestion d'un cache
	mapping réalisé automatiquement ou à l'aide d'un fichier de configuration au format XML
utilisation avec un SGBD uniquement	utilisation de tout type de format de stockage

JDO est une spécification : pour pouvoir l'utiliser il faut utiliser une implémentation fournie par un fournisseur.

Attention : tous les objets ne peuvent pas être rendu persistant avec JDO.

34.2. L'API JDO

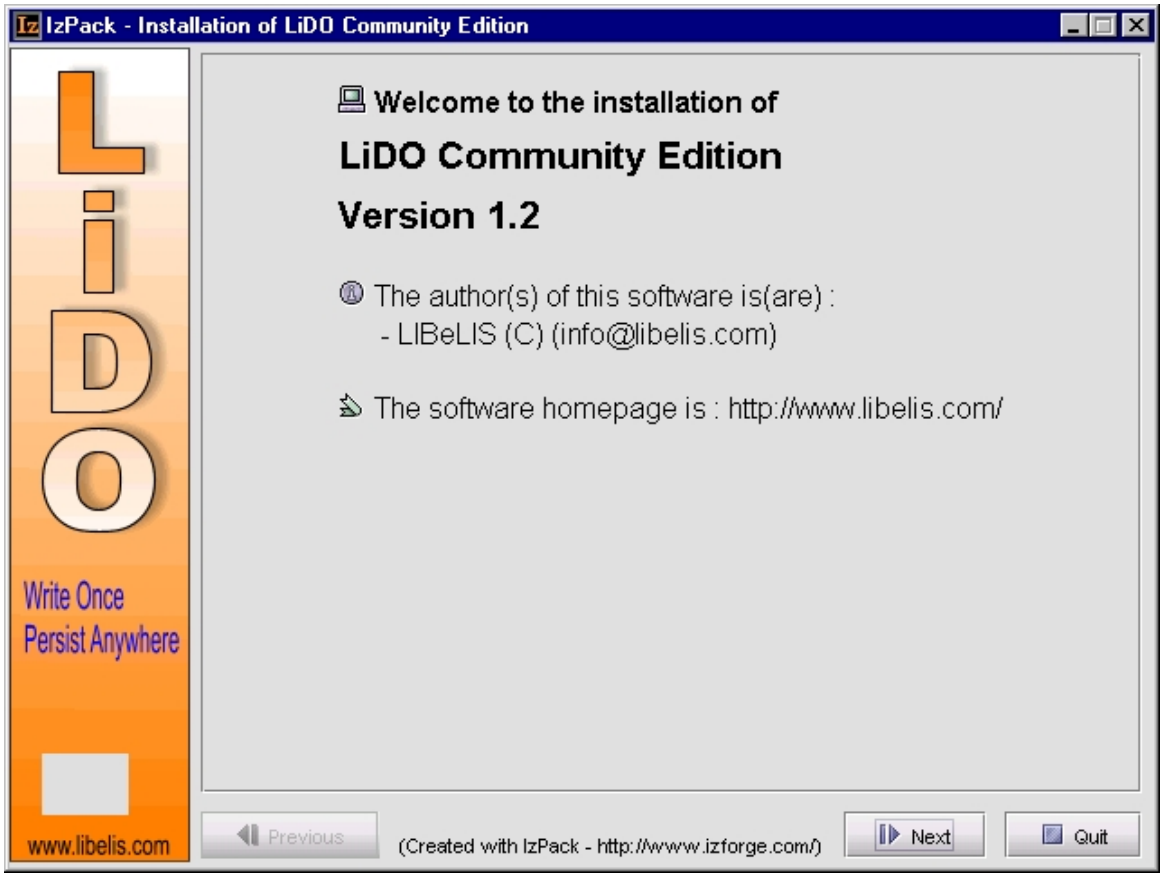
34.3. Un exemple avec lido

Les exemples de cette section on été réalisé avec lido community edition de la société Libelis.

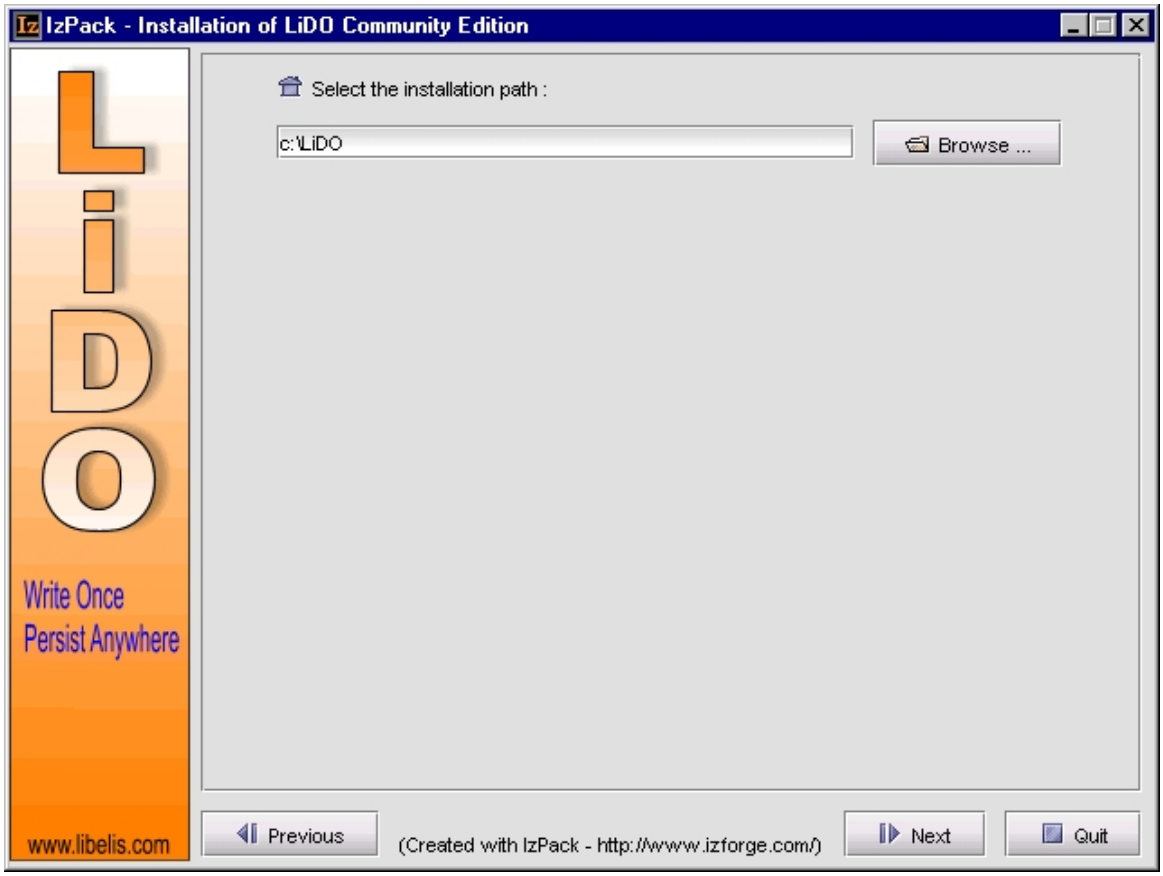
Cette version est librement téléchargeable après enregistrement à l'url : http://www.libelis.com

Pour lancer, l'installation il faut saisir la commande :

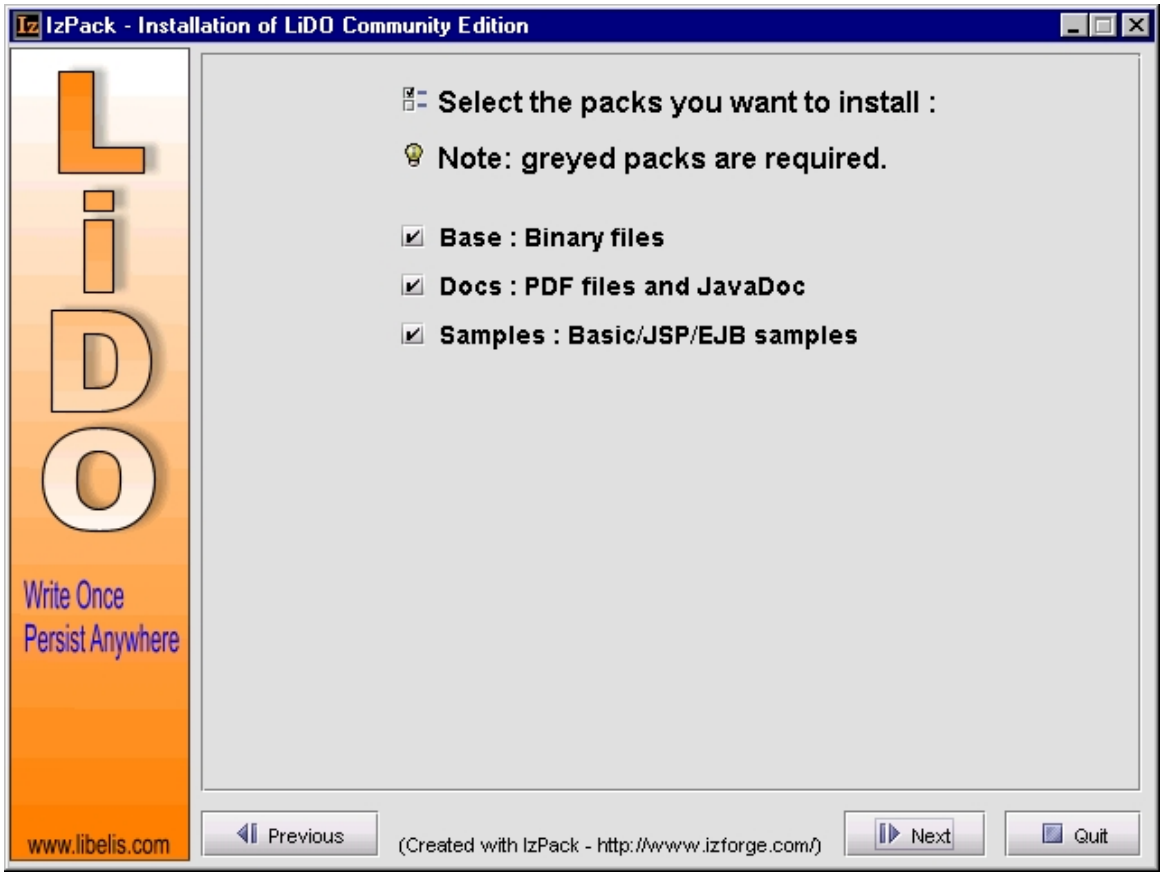
Installation de Lido community edition de Libelis
`java –jar lido_community_1[1].2.jar`



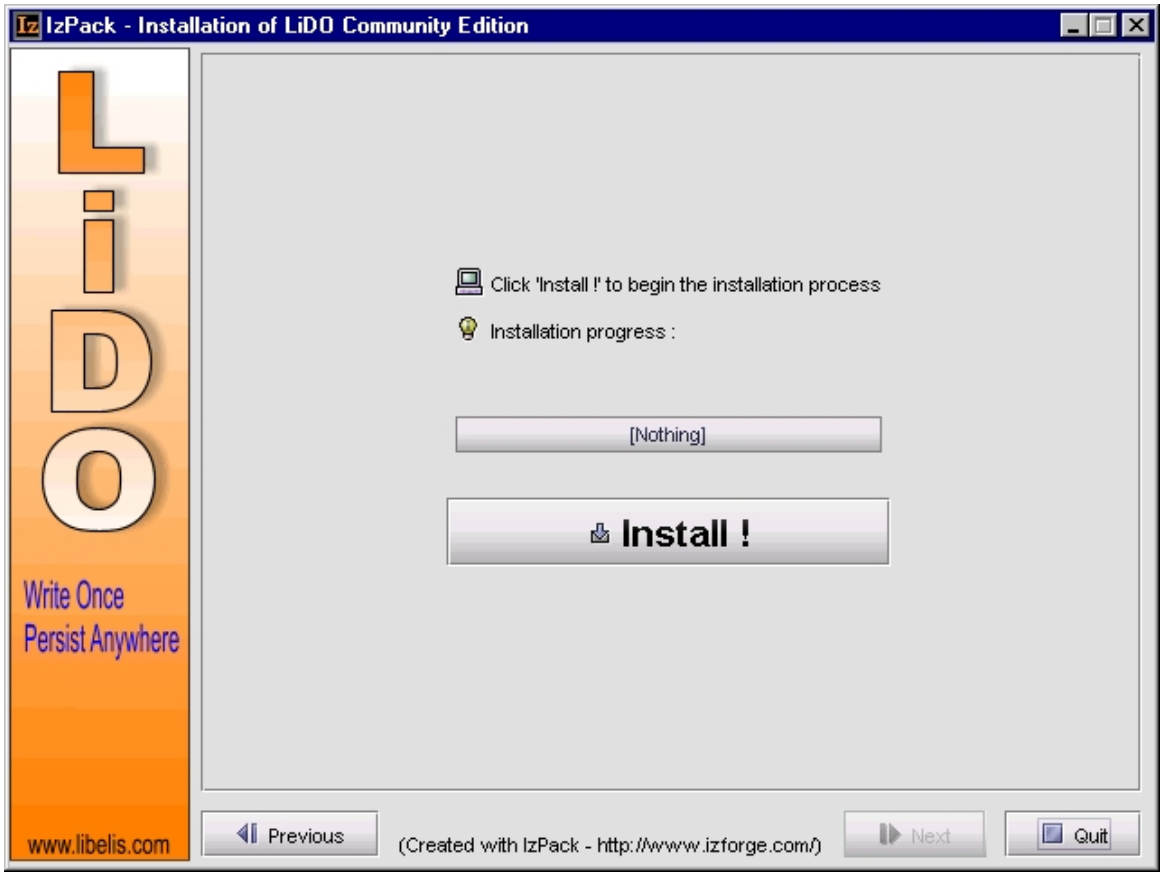
Après lecture et acceptations des conditions d'utilisation et de la license, il faut sélectionner le répertoire d'installation



Il choisir ensuite les composants à installer :



Puis valider l’installation en cliquant sur install



34.3.1. La création de l'objet qui va contenir les données

Le code de cette objet reste très basic puisque c'est un bean contenant des attributs.

Exemple :

```
package testjdo;

import java.util.*;

public class Personne {
      private String nom ="";
      private String prenom = "";
      private Date datenaiss = null;

      public Personne(String pNom, String pPrenom,Date pDatenaiss) {
            nom=pNom;
            prenom=pPrenom;
            datenaiss=pDatenaiss;
      }

      public String getNom() { return nom; }

      public String getPrenom() { return prenom;}

      public Date getDatenaisse() { return datenaiss; }

      public void setNom(String pNom) { nom = pNom; }

      public void setPrenom(String pPrenom) { nom = pPrenom; }

      public void setDatenaiss(Date pDatenaiss) { datenaiss = pDatenaiss; }

}
```

34.3.2. La création de l'objet qui sa assurer les actions sur les données

Cet objet va utiliser des objets jdo pour réaliser les actions sur les données. Dans l'exemple ci dessous, une seule action est codée : l'enregistrement dans la table des données de l'objet Personne.

Exemple :

```
package testjdo;

import javax.jdo.*;
import java.util.*;

public class PersonnePersist {

      private PersistenceManagerFactory pmf = null;
      private PersistenceManager        pm = null;
      private Transaction               tx  = null;

      public PersonnePersist() {
            try {

            pmf = (PersistenceManagerFactory) (
            Class.forName("com.libelis.lido.PersistenceManagerFactory").newInstance());
            pmf.setConnectionDriverName("org.gjt.mm.mysql.Driver");
            pmf.setConnectionURL("jdbc:mysql://localhost/testjava");
      } catch (Exception e ){
            e.printStackTrace();
            }
      }

      public void enregistrer() {
            Personne p = new Personne("mon nom","mon prenom",new Date());
            pm = pmf.getPersistenceManager();
            tx = pm.currentTransaction();
            tx.begin();
            pm.makePersistent(p);
            tx.commit();
            pm.close();
      }

      public static void main(String args[]) {
            PersonnePersist pp = new PersonnePersist();
            pp.enregistrer();
      }
}
```

34.3.3. La compilation

Les deux classes définies ci dessus doivent être compilées normalement.

Exemple : script de compilation

```
@Echo OFF
javac −verbose −classpath mm.mysql−2.0.14−bin.jar;c:\lido\lib\j2ee.jar;.;.\testjdo;c:\lido\lib\
lido−dev.jar;c:\lido\lib\lido−rdb.jar;c:\lido\lib\lido−rt.jar;c:\lido\lib\lido.tasks;c:\lido\li
b\lido.tld;c:\lido\lib\skinlf.jar.\testjdo\*.java
```

34.3.4. La définition d'un fichier metadata

Le fichier metadata est un fichier au format XML qui précise le mapping à réaliser.

Exemple :

```
<?xml version="1.0"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="testjdo">
    <class name="Personne"identity-type="datastore">
    <field name="nom"/>
    <field name="prenom"/>
    <field name="datenaiss"/>
    </class>
  </package>
</jdo>
```

34.3.5. L'enrichissement des classes contenant des données

Pour assurer une bonne execution, il faut enrichir l'objet Personne de code pour assurer la persistance par jdo. Lido fournit un objet pour

Exemple :

```
set LIDO_HOME=c:\LiDO
SET PATH=%LIDO_HOME%\bin;%PATH%

java -cp mm.mysql-2.0.14-bin.jar;c:\lido\lib\j2ee.jar;.;%LIDO_HOME%\bin;%CLASSPATH%;c:\lido\lib
\lido-dev.jar;c:\lido\lib\lido-rdb.jar;c:\lido\lib\lido-rt.jar;c:\lido\lib\lido.tasks;c:\lido\l
ib\lido.tld;c:\lido\lib\skinlf.jar com.libelis.lido.Enhance -classpath c:\$user\testjdo -metada
ta testjava.jdo -verbose
```

Résultat :

```
LiDO Enhancer
LiDO 1.2 build #015 (07/06/2002)
(C) 2001 LIBeLIS
Using : testjava.jdo
Classpath : c:\$user\testjdo
Analysing testjdo.Personne
 MANAGED     DEFAULT FETCH GROUP     private String nom
 MANAGED     DEFAULT FETCH GROUP     private String prenom
 MANAGED     DEFAULT FETCH GROUP     private java.util.Date datenaiss
Processing testjdo.Personne

it took 4 seconds
```

Le fichier Personne.class est enrichi (sa taille passe de 867 octets à 9737 octets)

34.3.6. La définition du schema de la base de données

Lido fournit un outils qui permet de générer la base de données contenant les tables pour le mapping des données.

Exemple :

```
set LIDO_HOME=c:\LiDO
SET PATH=%LIDO_HOME%\bin;%PATH%

java -cp mm.mysql-2.0.14-bin.jar;c:\lido\lib\j2ee.jar;.;%LIDO_HOME%\bin;%CLASSPATH%;c:\lido\lib
\lido-dev.jar;c:\lido\lib\lido-rdb.jar;c:\lido\lib\lido-rt.jar;c:\lido\lib\lido.tasks;c:\lido\l
ib\lido.tld;c:\lido\lib\skinlf.jar com.libelis.lido.ds.jdbc.DefineSchema -d jdbc:mysql://localh
ost/testjava -driver org.gjt.mm.mysql.Driver -metadata testjava.jdo
```

Resultat :

```
DefineSchema for RDBMS
Create schema ... please wait ...
Schema created ...
All connections have been closed
All is completed
```

Il est facile de vérifier les traitements effectués.

Exemple :

```
C:\mysql\bin>mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5 to server version: 3.23.49-max

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use testjava
Database changed
mysql> show tables;
+--------------------+
| Tables_in_testjava |
+--------------------+
| lidoidmax          |
| lidoidtable        |
| personne           |
| t_personne         |
+--------------------+
4 rows in set (0.38 sec)

mysql> describe lidoidmax;
+----------+------------+------+-----+---------+-------+
| Field    | Type       | Null | Key | Default | Extra |
+----------+------------+------+-----+---------+-------+
| LIDOLAST | bigint(20) |      |     | 0       |       |
+----------+------------+------+-----+---------+-------+
1 row in set (0.11 sec)

mysql> describe lidoidtable;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| LIDOID   | bigint(20)   |      | PRI | 0       |       |
| LIDOTYPE | varchar(255) |      | MUL |         |       |
+----------+--------------+------+-----+---------+-------+
2 rows in set (0.00 sec)

mysql> describe t_personne;
+-----------+---------------+------+-----+---------+-------+
| Field     | Type          | Null | Key | Default | Extra |
+-----------+---------------+------+-----+---------+-------+
| LIDOID    | bigint(20)    | YES  | MUL | NULL    |       |
| nom       | varchar(255)  | YES  |     | NULL    |       |
| prenom    | varchar(255)  | YES  |     | NULL    |       |
| datenaiss | timestamp(14) | YES  |     | NULL    |       |
+-----------+---------------+------+-----+---------+-------+
4 rows in set (0.05 sec)

mysql> select * from t_personne;
Empty set (0.39 sec)
```

34.3.7. L’execution de l’exemple

Exemple :

```
set LIDO_HOME=c:\LiDO
SET PATH=%LIDO_HOME%\bin;%PATH%

java −cp mm.mysql−2.0.14−bin.jar;c:\lido\lib\j2ee.jar;.;%LIDO_HOME%\bin;%CLASSPATH%;c:\lido\lib
\lido−dev.jar;c:\lido\lib\lido−rdb.jar;c:\lido\lib\lido−rt.jar;c:\lido\lib\lido.tasks;c:\lido\l
ib\lido.tld;c:\lido\lib\skinlf.jar −Djdo.metadata=testjava.jdo testjdo.PersonnePersist
```

A l’issu de l’éxecution, un enresigtrement est créé dans la table qui mappe l’objet Personne.

Exemple :

```
mysql> select * fromt_personne;
+--------+---------+------------+----------------+
| LIDOID | nom     | prenom     | datenaiss      |
+--------+---------+------------+----------------+
|      1 | mon nom | mon prenom | 20020612090753 |
+--------+---------+------------+----------------+
1 row in set (0.05 sec)
```

35. Les EJB (Entreprise Java Bean)

Les Entreprise Java Bean ou EJB sont des composants serveurs donc non visuel qui respectent les spécifications d'un modèle édité par Sun.

Le respect de ces spécifications permet d'utiliser les EJB de façon indépendante du serveur.

Le but des EJB est de faciliter la création d'applications distribuées pour les entreprises.

Une des principales caractéristiques des EJB est de permettre aux développeurs de se concentrer sur les traitements orientés métiers car les EJB et l'environnement dans lequel ils s'exécutent prennent en charge un certains nombres de traitements tel que la gestion des transactions, la persistance des données, la sécurité, ...

Physiquement, un EJB est un ensemble d'au moins deux classes regroupées dans un module contenant un descripteur particulier.

Pour obtenir des informations complémentaires sur les EJB, il est possible de consulter le site de Sun :
www.javasoft/products/ejb

Remarque : dans ce chapitre, le mot bean sera utilisé comme synonyme de EJB.

Ce chapitre est en cours d'écriture

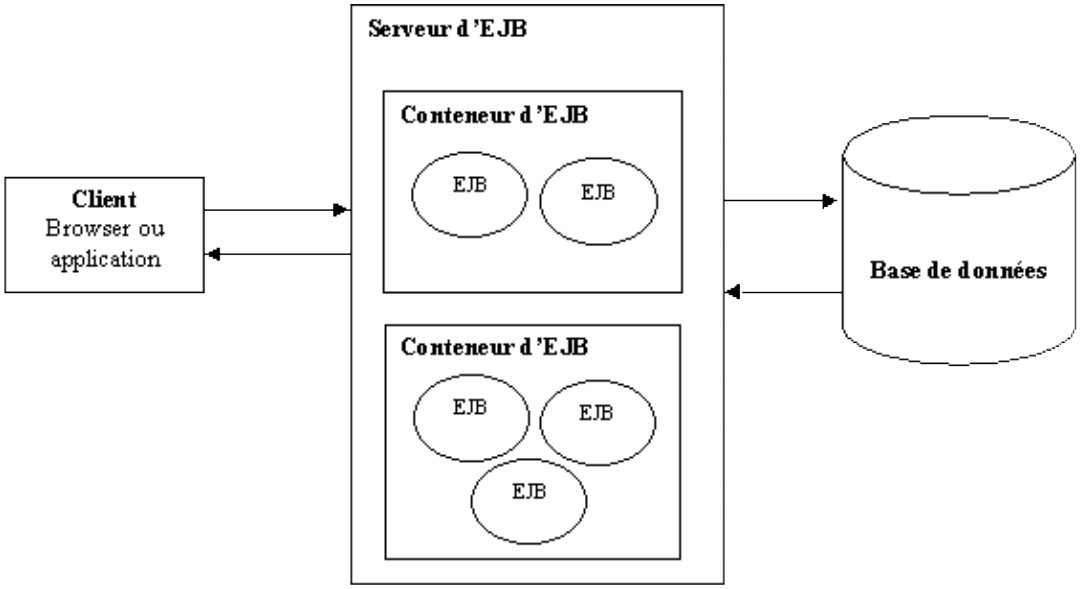
35.1. Présentation des EJB

Les EJB sont des composants et en tant que tel, ils possèdent certaines caractéristiques comme la réutilisabilité, la possibilité de s'assembler pour construire une application, etc ... Les EJB et les beans n'ont en communs que d'être des composants. Les java beans sont des composants qui peuvent être utilisés dans toutes les circonstances. Les EJB doivent obligatoirement s'éxécuter dans un environnement serveur dédié.

Les EJB sont parfaitement adaptés pour être intégrés dans une architecture trois tiers ou plus. Dans une telle architecture, chaque tiers assure une fonctionne particulière :

- le client « léger » assure la saisie et l'affichage des données
- sur le serveur, les objets métiers contiennent les traitements. Les EJB sont spécialement conçus pour constituer de tels entités.
- une base de données assure la persistance des informations

Les EJB s'exécutent dans un environnement particulier : le serveur d'EJB. Celui ci fournit un ensemble de fonctionnalités utilisées par des conteneurs d'EJB qui constituent le serveur d'EJB. Un serveur peut contenir plusieurs conteneurs. En réalité, c'est dans un conteneur que s'exécute une EJB.



Il existe de nombreux serveurs d'EJB commerciaux : BEA Weblogic, IBM Webpshere, Sun IPlanet, Macromedia JRun, Borland AppServer, etc ... Il existe aussi des serveurs d'EJB open source dont le plus avancé est JBoss.

Les entités externes au serveur qui appellent un EJB ne communique pas directement avec celui ci. Un objet héritant de EJBObject assure le dialogue entre ces entités et les EJB.

35.1.1. Les différents types d'EJB

Il existe deux types d'EJB : les beans de session (session beans) et les beans entité (les entity beans). Depuis la version 2.0 des EJB, il existe un troisième type de bean : les beans orienté message (message driven bean). Ils possèdent des points communs : ils doivent être déployés dans un conteneur d'EJB lequel s'execute dans un serveur d'EJB.

Les session beans peuvent être de deux types : sans état (stateless) ou avec état (statefull).

Les beans de session sans état peuvent être utilisés pour traiter les requètes de plusieurs clients. Les beans de session avec état ne sont accessibles que lors d'un ou plusieurs échanges avec le client. Le bean peut conserver des données entre les échanges

Les beans entité assurent la persistance des données. Il existe deux types d'entity bean :

- persistance géré par le conteneur (CMP : Container managed persistence)
- persistance géré par le bean (BMP : bean managed persistence).

Avec un entity bean CMP (container−managed persistence), c'est le conteneur d'EJB qui assure la persistance des données.Un entity bean BMP (bean−managed persistence), assure lui même la persistance des données grâce à du code inclus dans le bean.

La spécification 2.0 des EJB définie un troisième type d'EJB : les beans orientés messages (message−driven beans).

35.1.2. Le développement d'un EJB

Le cycle de développement d'un EJB comprend :

- la création des classes du bean
- la packaging du bean sous forme de fichier archive jar
- le deploiement du bean dans le serveur d'EJB
- le test du bean

La création d’un bean nécessite la création d’au minimum 3 classes pour respecter les spécifications de Sun : la classe du bean, l’interface remote et l’interface home.

L’interface remote permet de définir l’ensemble des services fournis par le bean. Cette interface étend l’interface EJBObject.

L’interface home permet de définir l’ensemble des services qui vont gérer le cycle de vie du bean. Cette interface étend l’interface EJBHome.

La classe du bean contient les traitements du bean. Cette classe implémente les méthodes déclarées dans les interfaces home et remote. Les méthodes définissant celle de l’interface home sont obligatoirement préfixées par « ejb ».

L'accès aux fonctionnalités du bean se fait obligatoirement par les méthodes définies dans les interfaces home et remote.

35.1.3. L'interface remote

L'interface remote permet de définir les méthodes qui contiendront les traitements proposés par le bean. Cette interface doit étendre l'interface EJBObject.

Exemple :

```
import javax.ejb.EJBObjet;

import java.rmi.RemoteException;

public interface MonBeanRemote extends EJBObject {

public String maMethode() throws RemoteException;

}
```

35.1.4. L'interface home

L'interface home permet de définir des méthodes qui vont gérer le cycle de vie du bean. Cette interface doit étendre l'interface EJBHome.

La création d'une instance d'un bean se fait grâce à une ou plusieurs surchages de la méthode create(). Chacune de ces méthodes renvoie une instance d'un objet du type de l'interface remote.

Exemple :

```
import javax.ejb.EJBHome;

import java.rmi.RemoteException;

import javax.ejb.CreateException;

public interface MonBeanHome extends EJBHome {

public MonBeanRemote create() throws RemoteException, CreateException;

}
```

35.3. Les EJB session

Un EJB session est un EJB de service dont la durée de vie correspond à un échange avec un client. Ils contiennent les règles métiers de l'application.

Ce type de bean implémente l'interface SessionBean.

Il existe deux types de session bean : stateless et statefull

Les session bean statefull sont capables de conserver l'état du bean dans ses variables d'instance durant toute la conversation avec un client. Mais ces données ne sont pas persistantes : à la fin de l'échange avec le client, l'instance de l'EJB est détruite et les données sont perdues.

Les EJB session stateless ne peuvent pas conserver de telles données.

Il ne faut pas faire appel directement aux méthodes create() et remove() de l'EJB. C'est le conteneur d'EJB qui se charge de la gestion du cycle de vie de l'EJB et qui appelle ces méthodes.

35.4. Les EJB entity

Ces EJB permettent de représenter des données enregistrées dans une base de données. Ils implémentent l'interface EntityBean.

Les beans entité assurent la persistance des données en représentant tout au partie d'une table ou d'une vue.. Il existe deux types d'entity bean :

- persistance gérée par le conteneur (CMP : Container managed persistence)
- persistance gérée par le bean (BMP : bean managed persistence).

Avec un entity bean CMP (container−managed persistence), c'est le conteneur d'EJB qui assure la persistance des données grâce aux paramètres fournie dans le descripteur de déploiement du bean.

Un bean entitéBMP (bean−managed persistence), assure lui même la persistance des données grâce à du code inclus dans les méthodes du bean.

Plusieurs clients peuvent accéder simultanément à un même EJB entity.

La gestion des transactions et des accès concurents sont assurés par le conteneur.

35.5. Les outils pour développer et mettre œuvre des EJB

35.5.1. Les outils de développement

Plusieurs EDI (environnement de développement intégré) commerciaux fournissent dans leur version Entreprise des outils pour développer et tester des EJB. On peut citer Jbuilder d'Inprise/Borland ou Visual Age Java ou WSAD d'IBM. Mais ces produits sont très coûteux pour une utilisation personnelle.

35.5.2. Les serveurs d’EJB

35.5.2.1. Jboss

JBoss est un serveur d’EJB Open Source écrit en java.

Il peut être téléchargé sur www.jboss.org.

JBoss nécessite la présence du J.D.K. 1.3.

Pour l’installer, il suffit de dézipper l’archive et de copier son contenu dans un répertoire , par exemple : c\jboss

Pour lancer le serveur, il suffit d’executer la commande :

java –jar run.jar

Les EJB à déployer doivent être mis dans le répertoire deploy. Si le répertoire existe au lancement du serveur, les EJB seront automatiquement deployés dès qu’ils seront insérés dans ce répertoire.

35.6. Le déploiement des EJB

Pour permettre le déploiement d’un EJB, il faut définir un fichier DD (deployement descriptor) qui contient des informations sur le bean.Ce fichier au format XML permet de donner au conteneur d'EJB des caractéristiques du bean.

Un EJB doit être déployé sous forme d’archive jar.

36. Les services web

Ce chapitre est en cours d'écriture

Partie 5 : les outils pour le développement

Le développement dans n'importe quel langage nécessite un ou plusieurs outils. D'ailleurs la multitude des technologies mises en oeuvre dans les projets récents nécessitent l'usage de nombreux outils.

Ce chapitre propose un recensement non exhaustif des outils utilisables pour le développement d'applications java et une présentation détaillée de certains d'entre eux.

Le JDK fournie un ensemble d'outils pour réaliser les développements mais leurs fonctionnalités se veulent volontairement limitées au strict miminum.

Enfin pour faciliter le developpement d'applications, il est préférable d'utiliser une méthodologie pour l'analyse et d'utiliser ou de définir des normes lors du développement.

Cette partie contient plusieurs chapitres :

- les outils du JDK : indique comment utiliser les outils fournis avec le JDK
- les outils libres et commerciaux : tente une énumération non exhaustive des outils libres et commerciaux pour utiliser java
- javadoc : explore l'outils de documentation fourni avec le JDK
- java et UML :
- normes de développement : propose de sensibiliser le lecteur à l'importance de la mise en place de normes de développement sur un projet et propose quelques règles pour définir une tel norme.
- les motifs de conception (design pattern) :

37. Les outils du J.D.K.

Le JDK de Sun fourni un ensemble d'outils qui permettent de réaliser des applications. Ces outils sont peu ergonomiques car ils s'utilisent en ligne de commande mais en contre partie ils peuvent toujours être utilisées.

37.1. Le compilateur javac

Cet outils est le compilateur : il utilise un fichier source java fourni en paramètre pour créer un ou plusieurs fichiers contenant le byte code java correspondant. Pour chaque fichier source, un fichier portant portant le même nom avec l'extension .class est créé si la compilation se déroule bien. Il est possible qu'un ou plusieurs autres fichier .class soit générés lors de la compilation de la classe si celle ci contient des classes internes. Dans ce cas, le nom du fichier des classes internes est de la forme classe$classe_interne.class. Un fichier .class supplémentaire est créé pour chaque classe interne.

37.1.1. La syntaxe de javac

La syntaxe est la suivante :

javac [options] [fichiers] [@fichiers]

Cet outils est disponible depuis le JDK 1.0

La commande attend au moins un nom de fichier contenant du code source java. Il peut y en avoir plusieurs, en les précisant un par un séparé par un escape ou en utilisant les jokers du système d'exploitation. Tous les fichiers précisés doivent obligatoirement posséder l'extension .java qui doit être précisée sur la ligne de commande.

Exemple : pour compiler le fichier MaClasse.

javac MaClasse.java

Exemple : pour compiler tous les fichier sources du répertoire

javac *.java

Le nom du fichier doit correspondre au nom de la classe contenue dans le fichier source. Il est obligatoire de respecter la casse du nom de la classe même sur des systèmes qui ne sont pas sensibles à la classe comme Windows.

Depuis le JDK 1.2, il est aussi possible de fournir un ou plusieurs fichiers qui contiennent une liste des fichiers à compiler. Chacun des fichiers à compiler doit être sur une ligne distincte. Sur la ligne de commande, les fichiers qui contiennent une liste doivent être précédés d'un caractère @

Exemple :

javac @liste

Contenu du fichier liste :
test1.java
test2.java

37.1.2. Les options de javac

Les principales options sont :

Option	Rôle
−classpath path	permet de préciser le chemin de recherche des classes nécessaire à la compilation
−d répertoire	les fichiers sont créés dans le répertoire indiqué. Par défaut, les fichiers sont créés dans le même répertoire que leurs sources.
−g	génère des informations débogage
−nowarn	le compilateur n'émets aucun message d'avertissement
−O	le compilteur procède à quelques optimisations. La taille du fichier généré peut augmenter. Il ne faut pas utiliser cette option avec l'option −g
−verbose	le compilateur affiche des informations sur les fichiers sources traités et les classes chargées
−deprecation	donne des informations sur les méthodes dépreciées qui sont utilisées

37.2. L'interpréteur java/javaw

Ces deux outils sont les interpréteurs de byte code : ils lancent le JRE, chargent les classes nécessaires et executent la méthode main de la classe.

java ouvre une console pour recevoir les messages de l'application alors que javaw n'en n'ouvre pas.

37.2.1. La syntaxe de l'outils java

java [options] classe [argument ...]
java [options] −jar fichier.jar [argument ...]
javaw [options] classe [argument ...]
javaw [options] −jar fichier.jar [argument ...]

classe être doit un fichier .class dont il ne faut pas préciser l'extension. La classe contenue dans ce fichier doit obligatoirement contenir une méthode main(). La casse du nom du fichier doit être respectée.

Cet outils est disponible depuis la version 1.0 du JDK.

Exemple:

java MaClasse

Il est possible de fournir des arguments à l'application.

37.2.2. Les options de l'outils java

Les principales options sont :

Option	Rôle
−jar archive	Permet d'exécuter une application contenue dans un fichier .jar. Depuis le JDK 1.2
−Dpropriete=valeur	Permet de définir une propriété système sous la forme propriete=valeur. propriétéreprésente le nom de la propriété et valeur représente sa valeur. Il ne doit pas y avoir d'espace entre l'option et la définition ni dans la définition. Il faut utiliser autant d'option −D que de propriétés à définir. Depuis le JDK 1.1
−classpath chemins ou −cp chemins	permet d'indiquer les chemins de recherche des classes nécessaires à l'execution. Chaque répertoire doit être séparé avec un point virgule. Cette option utilisée annule l'utilisation de la variable système CLASSPATH
−classic	Permet de préciser que c'est la machine virtuelle classique qui doit être utilisée. Par défaut, c'est la machine virtuelle utilisant la technologie HotSpot qui est utilisée. Depuis le JDK 1.3
−version	Affiche des informations sur l'interpréteur
−verbose ou −v	Permet d'afficher chaque classe chargée par l'interpréteur
−X	Permet de préciser des paramètres particuliers à l'interpréteur. Depuis le JDK 1.2

L'option −jar permet d'executer une application incluses dans une archive jar. Dans ce cas, le fichier manifest de l'archive doit préciser qu'elle est la classe qui contient la méthode main().

37.3. L'outil JAR

JAR est le diminutif de Java ARchive. C'est un format de fichier qui permet de regrouper des fichiers contenant du byte−code java (fichier .class) ou des données utilisées en temps que ressources (images, son, ...). Ce format est compatible avec le format ZIP : les fichiers contenus dans un jar sont compressés de façon indépendante du système d'exploitation.

Les jar sont utilisables depuis la version 1.1 du JDK.

37.3.1. L'intérêt du format jar

Leur utilisation est particulièrement pertinente avec les applets, les beans et même les applications. En fait, le format jar est le format de diffusion de composants java.

Les fichiers jar sont par défaut compressés ce qui est particulièrement intéressant quelque soit leurs utilisations.

Pour une applet, le browser n'effectue plus qu'une requête pour obtenir l'applet et ses ressources au lieu de plusieurs pour obtenir tous les fichiers nécessaires (fichiers .class, images, sons ...).

Un jar peut être signé ce qui permet d'assouplir et d'élargir le modèle de sécurité, notamment des applets qui ont des droits restreints par défaut.

Les beans doivent obligatoirement être diffusés sous ce format.

Les applications sous forme de jar peuvent être exécuter automatiquement.

Une archive jar contient un fichier manifest qui permet de préciser le contenu du jar et de fournir des informations sur celui ci (classe principale, type de composants, signature ...).

37.3.2. La syntaxe de l'outil jar

Le JDK fourni un outil pour créer des archives jar : jar. C'est un outil utilisable avec la ligne de commandes comme tous les outils du JDK.

La syntaxe est la suivante :

jar [option [jar [manifest [fichier

Cet outil est disponible depuis la version 1.1 du JDK.

Les options sont :

Option	Rôle
c	Création d'un nouvelle archive
t	Affiche le contenu de l'archive sur la sortie standard
x	Extraction du contenu de l'archive
u	Mise à jour ou ajout de fichiers à l'archive : à partir de java 1.2
f	Indique que le nom du fichier contenant l'archive est fourni en paramètre
m	Indique que le fichier manifest est fourni en paramètre
v	Mode verbeux pour avoir des informations complémentaires
0 (zéro)	Empêche la compression à la création
M	Empêche la création automatique du fichier manifest

Pour fournir des options à l'outils jar, il faut les saisir sans '−' et les accoler les uns aux autres. Leur ordre n'a pas d'importance.

Une restriction importante concerne l'utilisation simultanée du paramère 'm' et 'f' qui nécessite respectivement le nom du fichier manifest et le nom du fichier archive en paramètre de la commande. L'ordre des ces deux paramètres doit être identique à l'ordre des paramètres 'm' et 'f' sinon une exception est levée lors de l'exécution de la commande

Exemple (code java 1.1) :

```
C:\jumbo\Java\xagbuilder>jar cmf test.jar manif.mf *.class
java.io.IOException: invalid header field
        at java.util.jar.Attributes.read(Attributes.java:354)
        at java.util.jar.Manifest.read(Manifest.java:161)
        at java.util.jar.Manifest.<init>(Manifest.java:56)
        at sun.tools.jar.Main.run(Main.java:125)
        at sun.tools.jar.Main.main(Main.java:904)
```

Voici quelques exemples de l'utilisation courante de l'outils jar :

- Création d'un jar avec un fichier manifest créé automatiquement contenant tout les fichiers .class du répertoire courant

  jar cf test.jar *.class

- lister le contenu d'un jar

  jar tf test.jar

- Extraire le contenu d'une archive

  jar xf test.jar

37.3.3. La création d'une archive jar

L'option 'c' permet de créer une archive jar. Par défaut, le fichier créé est envoyé sur la sortie standard sauf si l'option 'f' est utilisée. Elle précise que le nom du fichier est fourni en paramètre. Par convention, ce fichier a pour extension .jar.

Si le fichier manifest n'est pas fourni, un fichier est créé par défaut dans l'archive jar dans le répertoire META−INF sous le nom MANIFEST.MF

Exemple (code java 1.1) : Création d'un jar avec un fichier manifest créé automatiquement contenant tout les fichiers .class du répertoire courant

```
jar cf test.jar *.class
```

Il est possible d'ajouter des fichiers contenus dans des sous répertoires du répertoire courant : dans ce cas, l'arborescence des fichiers est conservée dans l'archive.

Exemple (code java 1.1) : Création d'un jar avec un fichier manifest fourni contenant tous les fichiers .class du répertoire courant et tous les fichiers du répertoire images

```
jar cfm test.jar manifest.mf .class images
```

Exemple (code java 1.1) : Création d'un jar avec un fichier manifest fourni contenant tous les fichiers .class du répertoire courant et tous les fichiers .gif du répertoire images

```
jar cfm test.jar manifest.mf *.class images/*.gif
```

37.3.4. Lister le contenu d'une archive jar

L'option 't' permet de donner le contenu d'une archive jar.

Exemple (code java 1.1) : lister le contenu d'une archive jar

```
jar tf test.jar
```

Le séparateur des chemins des fichiers est toujours un slash quelque soit la plate−forme car le format jar est indépendant de toute plate−forme. Les chemins sont toujours donnés dans un format relatif et non pas absolu : le chemin est donné par rapport au répertoire courant. Il faut en tenir compte lors d'une extraction.

Exemple (code java 1.1) :

```
C:\jumbo\bin\test\java>jar tvf test.jar
  2156 Thu Mar 30 18:10:34 CEST 2000 META-INF/MANIFEST.MF
   678 Thu Mar 23 12:30:00 CET 2000   BDD_confirm$1.class
   678 Thu Mar 23 12:30:00 CET 2000   BDD_confirm$2.class
  4635 Thu Mar 23 12:30:00 CET 2000   BDD_confirm.class
   658 Thu Mar 23 13:18:00 CET 2000   BDD_demande$1.class
   657 Thu Mar 23 13:18:00 CET 2000   BDD_demande$2.class
   662 Thu Mar 23 13:18:00 CET 2000   BDD_demande$3.class
   658 Thu Mar 23 13:18:00 CET 2000   BDD_demande$4.class
  5238 Thu Mar 23 13:18:00 CET 2000   BDD_demande.class
   649 Thu Mar 23 12:31:28 CET 2000   BDD_resultat$1.class
  4138 Thu Mar 23 12:31:28 CET 2000   BDD_resultat.class
   533 Thu Mar 23 13:38:28 CET 2000   Frame1$1.class
   569 Thu Mar 23 13:38:28 CET 2000   Frame1$2.class
   569 Thu Mar 23 13:38:28 CET 2000   Frame1$3.class
  2150 Thu Mar 23 13:38:28 CET 2000   Frame1.class
   919 Thu Mar 23 12:29:56 CET 2000   Test2.class
```

37.3.5. L'extraction du contenu d'une archive jar

L'option 'x' permet d'extraire par défaut tout les fichiers contenu dans l'archive dans le répertoire courant en respectant l'arborescence de l'archive. Pour n'extraire que certains fichiers de l'archive, il suffit de les préciser en tant que paramètres de l'outils jar en les séparant par un espace. Pour une extraction totale ou partielle de l'archive, les fichiers sont extraits en conservant la hiérarchie des répertoires qui les contiennent.

Exemple (code java 1.1) : Extraire le contenu d'une archive

```
jar xf test.jar
```

Exemple (code java 1.1) : Extraire les fichiers test1.class et test2.class d'une archive

```
jar xf test.jar test1.class test2.class
```

Attention : lors de l'extraction, l'outils jar écrase tous les fichiers existants sans demander de confirmation.

37.3.6. L'utilisation des archives jar

Dans une page HTML, pour utiliser une applet fournie sous forme de jar, il faut utiliser l'option archive du tag applet. Cette option attend en paramètre le fichier jar et son chemin relatif par rapport au répertoire contenant le fichier HTML.

Exemple (code java 1.1) : le fichier HTML et le fichier MonApplet.jar sont dans le même répertoire

```
<applet code=MonApplet.class
        archive="MonApplet.jar"
        width=300 height=200>
</applet>
```

Avec java 1.1, l'exécution d'une application sous forme de jar se fait grâce au jre. Il faut fournir dans ce cas le nom du fichier jar et le nom de la classe principale.

Exemple (code java 1.1) :

```
jre -cp MonApplication.jar ClassePrincipale
```

Avec java 1.2, l'exécution d'une application sous forme de jar impose de définir la classe principale (celle qui contient la méthode main) dans l'option Main-Class du fichier manifest. Avec cette condition l'option -jar de la commande java

permet d'exécuter l'application.

Exemple (code java 1.2) :

```
java −jar MonApplication.jar
```

37.3.7. Le fichier manifest

Le fichier manifest contient de nombreuses informations sur l'archive et son contenu. Ce fichier est le support de toutes les fonctionnalités particulière qui peuvent être mise en oeuuvre avec une archive jar.

Dans une archive jar, il ne peut y avoir qu'un seul fichier manifest nommé MANIFEST dans le répertoire META−INF de l'archive.

Le format de ce fichier est de la forme clé/valeur. Il faut mettre un ':' et un espace entre la clé et la valeur.

```
C:\jumbo\bin\test\java>jar xf test.jar META−INF/MANIFEST.MF
```

Cela créé un répertoire META−INF dans le répertoire courant contenant le fichier MANIFEST.MF

Exemple (code java 1.1) :

```
Manifest−Version: 1.0
Name: BDD_confirm$1.class
Digest−Algorithms: SHA MD5
SHA−Digest: ntbIs5E5YNilE4mf570JoIF9akU=
MD5−Digest: R3zH0+m9lTFq+BlQvfQdHA==
Name: BDD_confirm$2.class
Digest−Algorithms: SHA MD5
SHA−Digest: 3QEF8/zmiTAP7MHFPU5wZyg9uxc=
MD5−Digest: swBXXptrLLwPMw/bpt6F0Q==
Name: BDD_confirm.class
Digest−Algorithms: SHA MD5
SHA−Digest: pZBT/o8YeDG4q+XrHRgrB08k4HY=
MD5−Digest: VFvY4sGRfjV1ciM9C+QIdg==
```

Dans le fichier manifest créé automatiquement avec le JDK 1.1, chaque fichier possède au moins une entrée de type 'Name' et des informations les concernant.

Entre les données de deux fichiers, il y a une ligne blanche.

Dans le fichier manifest créé automatiquement avec le JDK 1.2, il n'y a plus d'entrée pour chaque fichier.

Exemple (code java 1.1) :

```
Manifest−Version: 1.0
Created−By: 1.3.0 (Sun Microsystems Inc.)
```

Le fichier manifest généré automatiquement convient parfaitement si l'archive est utilisée uniquement pour regrouper les fichiers. Pour une utilisation plus spécifique, il faut modifier ce fichier pour ajouter les informations utiles.

Par exemple, pour une application exécutable (à partir de java 1.2) il faut ajouter une clé Main−Class en lui associant le nom de la classe dans l'archive qui contient la méthode main.

37.3.8. La signature d'une archive jar

La signature d'une archive jar joue un rôle important dans les processus de sécurité de java. La signature d'une archive permet à celui qui utilise cette archive de lui donner des droits étendus une fois que la signature a été reconnue.

Avec Java 1.1 une archive signée possède tous les droits.

Avec Java 1.2 une archive signée peut se voir attribuer des droits particuliers définis un fichier policy.

37.4. Pour tester les applets : l'outil appletviewer

Cet outils permet de tester une applet. L'intérêt de cet outils est qu'il permet de tester une applet avec la version courante du JDK. Un navigateur classique nécessite un plug−in pour utiliser une version particulière du JRE. Cet outils est disponible depuis la version 1.0 du JDK.

En contre partie, l'appletviewer n'est pas prévu pour tester les pages HTML. Il charge une page HTML fournie en paramètre, l'analyse, charge l'applet qu'elle contient et execute cet applet.

La syntaxe est la suivante :

appletviewer [option fichier

L'appletview recherche le tag HTML <APPLET>. A partir du JDK 1.2, il recherche aussi les tags HTML <EMBED> et <OBJECT>.

Il possède plusieurs options dont les principales sont :

Option	Rôle
−J	Permet de passer un paramètre à la JVM. Pour passer plusieurs paramètres, il faut utiliser plusieurs options −J. Depuis le JDK 1.1
−encoding	Permet de préciser le jeu de caractères de la page HTML

L'appletviewer ouvre une fenêtre qui possède un menu avec les options suivante :

Option de menu	Rôle
Restart	Permet d'arrêter et de redémarrer l'applet
Reload	Permet d'arrêter et de recharger l'applet
Stop	Permet d'arrêter l'execution de l'applet. Depuis le JDK 1.1
Save	Permet de sauvegarder l'applet en la serialisant dans un fichier applet.ser. Il est necessaire d'arrêter l'applet avant d'utiliser cet option. Depuis le JDK 1.1
Start	Permet de démarrer l'applet. Depuis le JDK 1.1
Info	Permet d'afficher les informations de l'applet dans une boite de dialogue. Ces informations sont obtenues par les méthodes getAppletInfo() et getParameterInfo() de l'applet.

Print	Permet d'imprimer l'applet. Depuis le JDK 1.1
Close	Permet de fermer la fenêtre courante
Quit	Permet de fermer toutes les fenêtres ouvertes par l'appletviewer

37.5. Pour générer la documentation : l'outil javadoc

Cet outils permet de générer une documentation à partir des données insérées dans le code source.

37.5.1. La syntaxe de javadoc

La syntaxe est la suivante :

javac [options] [fichiers] [@fichiers]

Cet outils est disponible depuis le JDK 1.0

37.5.2. Les options de javadoc

Les principales options sont :

Option	Rôle
−1.1	Permet de générer une documentation au format défini par le JDK 1.1
−author	Permet d'inclure dans la documentation les données du tag @author
−d	Permet de préciser le répertoire qui va contenir les fichiers générés
−nodeprecated	Permet d'omettre les informations sur les éléments deprecated
−nodeprecatedlist	Permet de ne pas générer la page contenant les éléments deprecated
−noindex	Permet de ne pas générer la page index
−notree	Permet de ne pas générer la page contenant la hierachie de classes
−private	Permet d'inclure les éléments déclarés private
−protected	Permet d'inclute les éléments déclarés protected
−public	Permet de n'inclure que les éléments déclarés public
−verbose	Permet de fournir des informations lors de la génération

Des informations supplémentaires sur les éléments à inclure dans le code source sont fournies dans le chapitre Javadoc

38. Les outils libres et commerciaux

Pour développer des composants en java (applications clientes, applets, applications web, services web ...), il existe une large gamme d'outils commerciaux et libres pour répondre à ce vaste marché.

Comme dans d'autres domaines, les avantages et les incovénients de ces outils sont semblables selon leur catégorie bien qu'ils ne puissent pas être complétement généralisés :

	Avantages	Inconvénient
Outils commerciaux	une meilleur ergonomie une hot line dédiée	le prix
Outils libres	la gratuité des mises à jour fréquentes (variable selon le projet)	pas de support officiel (aide communautaire via les forums)

Certains de ces outils libres n'ont que peu de choses à envier à certains de leurs homologues commerciaux : ainsi Tomcat du projet Jakarta est l'implémentation de référence pour ce qui concerne les servlets et les JSP.

Enfin certains éditeurs, surtout dans le domaine des IDE, proposent souvent une version limitée (dans les fonctionnalités ou dans le temps)) mais gratuite qui permet d'utiliser et d'évaluer le produit.

L'évolution des ces outils suit l'évolution du marché concernant java : développement d'applet (web client), d'application automone et C/S, et maintenant développement côté serveur (applications et services web).

La liste des produits de ce chapitre est loin d'être exhaustive mais représente les plus connus ou ceux que j'utilise.

38.1. Les environnements de développements intégrés (IDE)

Les environnements de développements intégrés regroupent dans un même outils la possibilité d'écrire du code source, de concevoir une application de façon visuelle par assemblage de beans, d'exécuter et de débugger le code.

D'une façon générale, ils sont tous très gourmands en ressources machines : un processeur rapide, 256 Mo de RAM pour être à l'aise ... En fait la plupart de ces outils sont partiellement ou totalement écrit en java.

Le choix d'un IDE doit tenir compte de plusieurs caracéristiques : ergonomie et convivialité pour faciliter l'utilisation, fonctionnalités de bases et avancées pour accroître la productivité, robustesse, support des standards ... Tous les éditeurs proposent une version libre qui permet d'évaluer leur produit.

38.1.1. Borland JBuilder

Borland est spécialisé depuis des années dans la création d'outils de développement possédant d'excellente réputation. Ainsi Jbuilder est un IDE ergonomique qui génère un code "propre", ce qui lui vaut d'être l'IDE java le plus utilisé dans le monde. Depuis sa version 3.5, JBuilder est écrit en java 2 se qui lui permet de s'executer sans difficulté sur plusieurs plateformes notamment Windows, Linux ou Solaris.

http://www.inprise.fr/produits/jbuilder/

Le produit dispose de nombreuses caractéristique qui facilite le travail du développeur : la technologie CodeInsight facilite grandement l'écriture du code dans l'éditeur, de nombreux assistants facilitent la génération de code ...

Il existe plusieurs éditions :

- foundation ou édition personnelle (depuis la version 5) :
- professionnelle
- entreprise

La version 5 intègre fortement XML et elle est très ouverte : par exemple la version entreprise intègre des outils libre (Xerces, Xalan, Tomcat), permet la création d'archive jar, war et ear, permet le déploiement d'application J2EE vers des serveurs d'application concurent tel que Weblogic ou Websphere et des applications avec java web start, peut travailler avec les principaux gestionnaires de versions (CSV, Source Safe, Clear Case) ...

38.1.2. IBM Visual Age for Java

IBM propose une famille d'outils pour différents langages dont une version dédiée à java.

Visual Age for Java (VAJ) est un outils novateur dans son ergonomie et son utilisation qui sont complètement différentes des autres EDI. Les débuts de son utilisation sont parfois déroutant mais une persévérence permet de révéler toute sa puissance.

http://www−4.ibm.com/software/ad/vajava/

La fenêtre principale (plan de travail) est séparée en deux parties :

- l'espace de travail : il contient et organise les différents éléments (projets, packages, classes, méthodes ...)
- le code source : si l'élement sélectionné dans l'espace de travail contient du code, il est visualisé et modifiable dans cette partie

Par défaut le code est éditable par méthode mais depuis la version 3.5, il est toutefois possible de visualiser le code source complet mais les opérations réalisables dans ce mode sont moins nombreuses.

VAJ possède plusieurs points forts : le regroupement de toutes les classes et leur organisation dans l'espace de travail, la compilation incrémentale à l'écriture et au débuggage, le travail collaboratif avec le contrôle de version dans un référentiel (repository). Tous ces points facilitent le développement de gros projets mais il n'est pas adapté pour le déploiement du code écrit : il faut utiliser un autre produit.

VAJ est un outils puissant particulièrement adapté aux utilisateurs chevronnés pour de gros projets.

38.1.3. IBM Websphere Studio Application Developper

Websphere Studio Application Developper (WSAD) représente le nouvel outils de développement d'application java/web d'IBM. Il représente une fusion de nombreuses fonctionnalités des outils Visual Age for Java et Websphere Studio. Le coeur de l'outils est composé par Websphere Studio Workbench dont une partie du code a été fournie à la communauté open source pour devenir le projet eclipse. Le but est de founir un framework commun pour un outils de développement modulaire.

http://www-4.ibm.com/software/ad/studioappdev/

L'avantage de cette modularité est de fournir dans un même outils des fonctionnalités qui nécessitaient jusqu'à présent l'usage de plusieurs outils dont l'inter-opérabilité n'était pas parfaite.

Pour le moment, WSAD version 4.0 est orienté developpement java/web : il ne permet pas de developpement d'applications graphiques en mode RAD.

38.1.4. Netbeans

Netbeans est un environnement de développement en java open source écrit en java. Le produit est composé d'une partie centrale à laquelle il est possible d'ajouter des modules tel que poseidon pour la création avec UML.

http://www.netbeans.org/

Netbeans est la base de l'IDE de Sun Forte for Java.

38.1.5. Sun Forte for java

Forte for java est basé sur Netbeans que Sun a racheté.

http://www.sun.com/forte/ffj

La version Community Edition de Forte for java est téléchargeable gratuitement.

38.1.6. JCreator

http://www.jcreator.com

Jcreator existe en deux version : la version "LE" est en freeware et la version "PRO" est en shareware. Il est particulièrement rapide car il est écrit en code natif.

38.1.7. Le projet Eclipse

Eclipse est un projet open source à l'origine développé par IBM pour ces futurs outils de développement et offert à la communauté. Le but est de fournir un outils modulaire capable non seulement de faire du développement en java mais aussi dans d'autres langage et d'autres activités. Cette polyvalence est liée au développement de modules réalisés par la communauté ou des entités commerciales.

http://www.eclipse.org/

L'espace de travail permet de voir des perspectives qui assurent une vision particulière d'un projet. Chaques perspectives contient des vues et des éditeurs qui pemettent de travailler sur une entité.

38.1.8. Webgain Visual Café

Webgain Studio propose un ensemble d'outils (Visual Café, Dreamwaever Ultradev, Top link, Structure Builder, Weblogic) pour la création d'applications e−business. Visual Café est l'IDE de développement en java.

http://www.webgain.com/

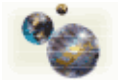
Visual Café existe en trois version : standard, expert et entreprise suite.

Il permet de travailler avec plusieurs JDK : 1.1, 1.2 et 1.3

38.2. Les serveurs d'application

Les serveurs d'applications sont des outils qui permettent l'execution de composants java côté serveur (servlets, JSP, EJB, ...).

38.2.1. IBM Websphere Application Server



Websphere Application Server (WAS) est le serveur d'application de la famille d'outils Websphere. Il permet de déploiement de composant java orienté web

http://www−4.ibm.com/software/webservers/

La version 4 est certifiée J2EE 1.2. Elle permet la mise en oeuvre des servlets, JSP, EJB et services Web (SOAP, UDDI, WSDL, XML). Cette version est proposée en 4 éditions qui supporte tout ou partie de ces composants :

- Standard Edition : pour les applications web utilisant des serlets, des JSP et XML
- Advanced Edition: supporte en plus les EJB, la répartition de charges sur plusieurs machines
- Advanced Single Server Edition : supporte toute les API J2EE mais uniquement sur une seule machine. Cette version ne peut pas être utilisée en production.
- Enterprise Edition : supporte en plus CORBA et la connection aux ressources de l'entreprise

38.2.2. BEA Weblogic

http://fr.bea.com/produits/weblogic.htm

38.2.3. iplanet

iPlanet™

http://fr.iplanet.com/

38.2.4. Borland Entreprise Server

http://www.borland.fr/produits/bes/

38.2.5. Macromedia Jrun Server

http://www.macromedia.com/software/jrun/

38.3. Les conteneurs web

Les conteneurs web sont des applications qui permettent d'executer du code java utilisé pour définir des servlets et des JSP.

38.3.1. Apache Tomcat

Tomcat est un conteneur d'applications web (servlets et JSP) développé par la fondation Apache. C'est l'implémentation de référence pour les API servlets et JSP : il est donc pleinement compatible avec les spécifications J2EE de ces API.

http://jakarta.apache.org/tomcat/

Les API supportés dépendent de la version du produit :

Version	API Servlets	API JSP
3.2	2.2	1.1
4	2.3	1.2

38.3.2. Caucho Resin

Resin est un moteur de servlet et de JSP qui intègre un serveur web.

http://www.caucho.com/

38.3.3. Enhydra

Enhydra est un projet open source, initialement créer par Lutris technologies , pour développer un conteneur web pour Servlets et JSP. Il fourni en plus quelques fonctionnalités supplémentaires pour utiliser XML, mapper des données avec des objets et gérer un pool de connection vers des bases de données

http://enhydra.enhydra.org/

38.4. Les conteneurs d'EJB

Les conteneurs d'EJB sont des applications qui fournissent un environnement d'execution pour les EJB.

38.4.1. JBoss

JBoss est un projet open source développé en java pour fournir un environnement d'execution d'EJB respectant les spécification J2EE.

http://www.jboss.org

JBoss est composé d'un ensemble d'outils : JBoss Server, JBoss MQ (implémentation de JMS), JBoss MX, JBoss TX (implémentation de JTA/JTS), JBoss SX , JBoss CX et JBoss CMP.

JBoss provides JBossServer, the basic EJB container and JMX infrastructure. JBossMQ, for JMS messaging, JBossMX, for mail, JBossTX, for JTA/JTS transactions, JBossSX for JAAS based security, JBossCX for JCA connectivity and JBossCMP for CMP persistence. JBoss enables you to mix and match these components through JMX by replacing ANY component you wish with a JMX compliant implementation for the same APIs. JBoss doesn't even impose the JBoss components, that is modularity.

38.4.2. Jonas

JOnAS est un projet open source développé en java visant a réaliser une implémentation des spécification EJB 1.1, JTA 1.0.1, JDBC 2.0 and JMS 2.0.1.

http://www.objectweb.org/jonas/index.html

38.4.3. OpenEJB

OpenEJB est un projet open source pour developper un contenur d'EJB qui respecte les spéifications 2.0 des EJB. Pour le moment, le projet est en cours de développement et il n'existe pas encore de binaires (seul les sources sont disponibles).

http://openejb.exolab.org/

38.5. Les outils divers

38.5.1. Jikes

Jikes est un compilateur Java open source écrit par IBM en code natif pour Windows et Linux. Son exécution est donc extrèmement rapide d'autant plus lorsqu'il s'agit de très gros projet sur une machine peu véloce.

http://www10.software.ibm.com/developerworks/opensource/jikes/

Pour utiliser Jikes, il sufiit de décompresser l'archive et de mettre le fichier exécutable dans un répertoire inclus dans le CLASSPATH. Enfin, il faut déclarer une varaible système JIKESPATH qui doit contenir les différents répertoire

contenant les classes et les jar notamment le fichier rt.jar du JRE.

38.5.2. GNU Compiler for Java

38.5.3. Argo UML

Argo UML est un projet open source écrit en java qui vise à developper un outils de modélisation UML 1.1. Il est possible de créer des diagrammes UML et de générer le code java correspondant au diagrammes de classes. Une option permet de créer les diagrammes de classes à partie du code source java

http://argouml.tigris.org/

Cet outils n'est pas encore en version finale mais la version 0.95 est très prometteuse.

38.5.4. Poseidon UML

http://www.gentleware.com/products/index.php3

38.5.5. Artistic Style

Artistic Style est un outils open source qui permet d'indenter et de formatter un code source C, C++ et java

http://astyle.sourceforge.net

Cet outils possèdent de nombreuses options de formattage de fichiers source. Les options les plus courantes pour un code source java sont :

astyle −jp −−style=java nomDuFichier.java

Par défaut, l'outils conserve le fichier original en le suffixant par .orig.

38.5.6. Ant

Ant est un outils du projet jakarta pour réaliser la compilation de projet java. C'est un équivalent à l'outils make sous Unix mais il est écrit en java et donc indépendant de toutes plateforme. Il permet donc la recompilation du projet sur toute plateforme équipée d'un JVM.

http://jakarta.apache.org/ant

Ant utilise un fichier de paramètres (buildfile) pour la compilation du projet au format XML.

38.5.7. Castor

http://castor.exolab.org/

38.5.8. Beanshell

http://www.beanshell.org/

38.5.9. Junit

http://www.junit.org/

38.6. Les MOM

Les Middleware Oriented Message sont des outils qui permettent l'échange de messages entre des composants d'une application ou entre applications. Pour pouvoir les utiliser avec java, ils doivent implémenter l'API JMS de Sun.

38.6.1. openJMS

http://openjms.exolab.org/

38.6.2. Joram

http://www.objectweb.org/joram/

39. JavaDoc

Javadoc est un outils fourni par Sun avec le JDK pour permettre la génération d'une documentation technique à partir du code source.

39.1. La documentation générée

Pour générer la documentation, il faut invoquer l'outils javadoc. Javadoc recrée à chaque utilisation la totalité de la documentation.

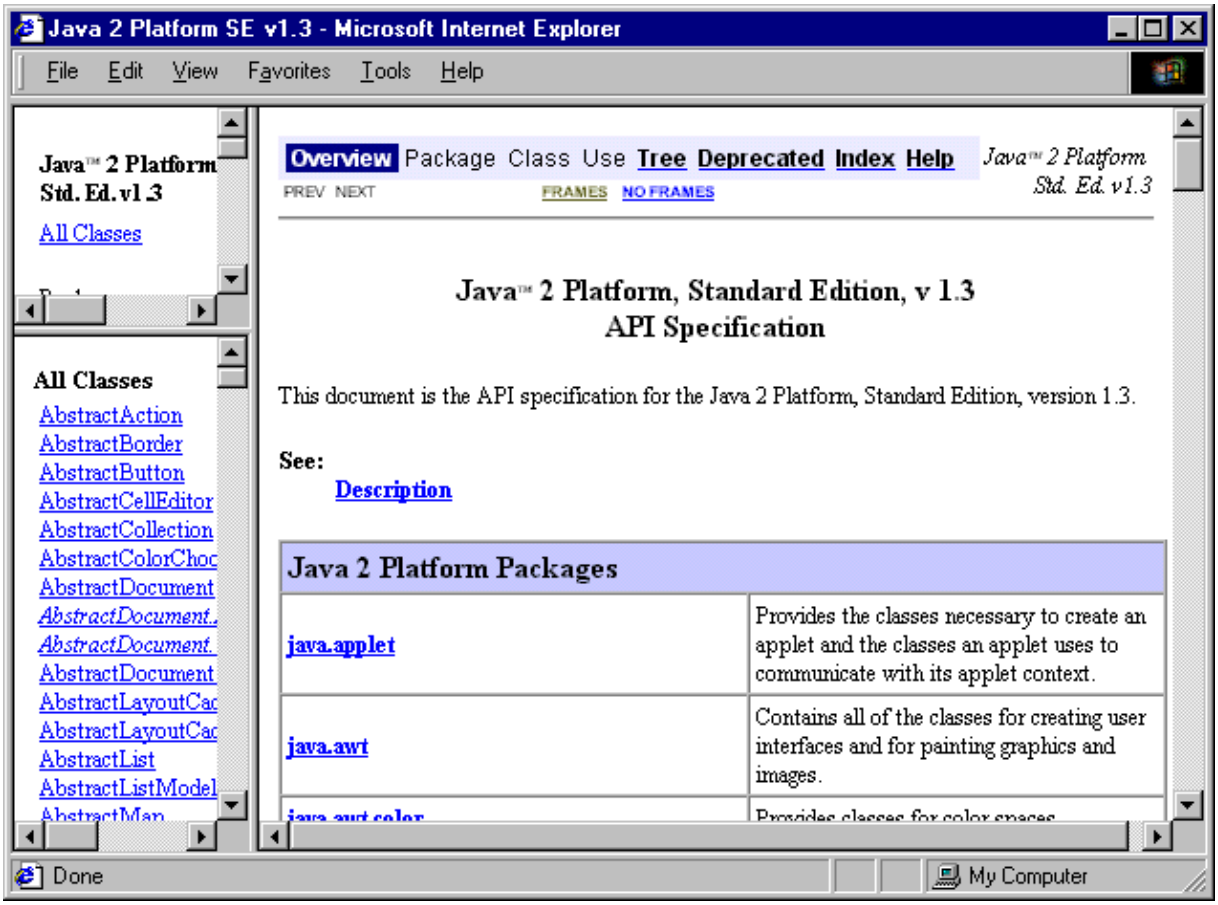
La documentation générée est par défaut au format HTML.

Pour formatter la documentation, javadoc utilise une doclet. Une doclet permet de préciser le format de la documentation générée. Par défaut, javadoc propose un doclet qui génère une documentation au format HTML. Il est possible de définir sa propre doclet pour changer le contenu ou le format de la documentation (pour par exemple, générer du RTF ou du XML).

Par défaut , la documentation générée contient les éléments suivants :

- un fichier html par classe ou interface qui contient le détail chaque élément de la classe ou interface
- un fichier html par package qui contient un résumé du contenu du package
- un fichier overview−summary.html
- un fichier overview−tree.html
- un fichier deprecated−list.html
- un fichier serialized−form.html
- un fichier overview−frame.html
- un fichier all−classe.html
- un fichier package−summary.html pour chaque package
- un fichier package−frame.html pour chaque package
- un fichier package−tree.html pour chaque package

La documentation de l'API java fourni par Sun est réalisée grâce à javadoc :



39.2. Les commentaires de documentation

Javadoc s'appuie sur le code source et sur un type de commentaires particuliers pour obtenir des données supplémentaires aux éléments qui composent le code source. Ces commentaires suivent des règles précises.

Ces commentaires commencent par /** et finissent par */ et contiennent toujours au minimum une phrase qui est un résumé de le l'élément. Si le commentaire utilise plusieurs ligne, javadoc ignore les premiers caractères d'espacement ainsi que le premier caractères * qui suit ces caractères. Ceci permet d'utiliser le caractère * pour aligner le contenu du commentaires

Il est possible de faire suivre cette suivre cette phrase d'un texte descriptif plus complet.

Il est possible d'utiliser des tags HTML pour formatter le texte : il ne faut pas utiliser de tags HTML de structure tel que Hn, HR ... qui sont utilisés par javadoc pour formatter la documentation.

Enfin il est possible d'utiliser des tags prédéfinis par javadoc pour fournir des informations plus précises sur des composants particuliers de l'élément (auteur, parametres, valeur de retour). Ces tags sont définis pour un ou plusieurs type d'élément.

Exemple (code java 1.0) :

```
/**
 * un commentaire javadoc
 */
```

Par défaut, javadoc prend en compte les éléments suivants : les classes, les interfaces, les méthodes et les champs public et protected. Le placement du commentaire de documentation dans le code source est important. Le documentaire est

associé à l'élément qui suit immédiatement le commentaire. Il faut ainsi pour faire précéder l'élément concerné par sa documentation et ne déclarer qu'une seule entité par ligne pour pouvoir lui associer la documentation.

39.3. Les tags définis par javadoc

Javadoc défini plusieurs tags qui permettent de précises certains composants de l'élément décrit de façon standardisée. Ces tags commencent tous par le caractère arobase @. Il existe deux types de tags :

- les tags standards : leur syntaxe est la suivante @tag
- les tags qui seront remplacés par une valeur : la syntaxe est la suivante {@tag}

Pour pouvoir être interprétés les tags standards doivent obligatoire commencé en début de ligne.

Tag	Rôle	élément concerné	version du JDK
@author	permet de préciser l'auteur de l'élement	classe et interface	1.0
@deprecated	permet de préciser qu'un élément est déprécié	package, classe, interface, méthode et champ	1.1
{@docRoot}	représente le chemin relatif du répertoire principal de génération de la documentation		1.3
@exception	permet de préciser une exception qui peut être levée par l'élément	méthode	1.0
{@link}	permet d'inserer un lien vers un élément de la documentation dans n'importe quel texte	package, classe, interface, methode, champ	1.2
@param	permet de préciser un paramètre de l'élément	constructeur et méthode	1.0
@see	permet de préciser un élément en relation avec l'élément documenté	package, classe, interface, champ	1.0
@serial		classe, interface	1.2
@serialData		methode	1.2
@serialField		classe, interface,	1.2
@since	permet de préciser depuis quel version l'élément aété ajouté	package, classe, interface, méthode et champ	1.1
@throws	identique à @exception	méthode	1.2
@version	permet de préciser le numéro de version de l'élément	classe et interface	1.0
@return	permet de préciser la valeur de retour d'un élément	méthode	1.0

39.3.1. Le tag @author

Ce tag permet de préciser le nom du ou des auteurs du code. Ce tag doit être utilisé uniquement pour un élement de type classe ou interface.

La syntaxe est la suivante :

@author nom_de_l_auteur

Ce tag génère une entrée Author: avec le nom de l'auteur dans la documentation. Par défaut, ce tag n'est pas pris en compte par javadoc. Pour qu'il soit pris en compte il faut utiliser l'option −author.

Pour préciser plusieurs noms, il faut les séparer par une virgule ou utiliser plusieurs tags chacun contenant un nom.

39.3.2. Le tag @deprecated

Ce tag permet de donner des précisions sur un élément déprécié (deprecated).

Ce tag doit être utilisé uniquement pour un élement de type classe ou interface.

La syntaxe est la suivante :

@deprecated explication

Il est utile de préciser depuis quelle version l'élément est déprécié et de préciser si un autre élément le remplace.

Ce tag génère une entrée Deprecated avec l'explication dans la documentation.

Ce tag est particulier car il est le seul reconnu par le compilateur : celui ci prend note de cet attribut lors de la compilation pour permettre d'informer les utilisateurs de cet élément.

39.3.3. La tag @exception

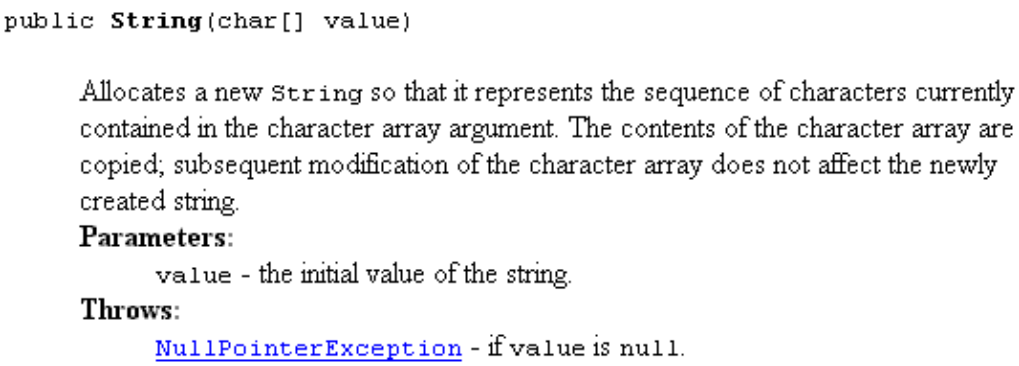
Ce tag permet de fournir des informations sur une exception qui peut être levée. Il faut le faire suivre du nom complétement qualifié de l'exception et d'une description des conditions de sa levée.

Ce tag doit être utilisé uniquement pour un élement de type méthode.

La syntaxe est la suivante :

@exception nom_de_la_classe description_de_l_exception

Exemple extrait de la documentation de l'API du JDK :



Il faut utiliser un tag @exception pour chaque exception déclarée dans la signature de la méthode.

39.3.4. Le tag @param

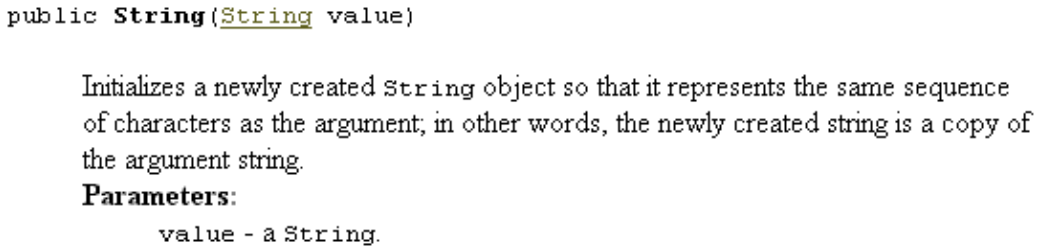
Ce tag permet de fournir des informations sur les paramètres. Ce tag doit être utilisé uniquement pour un élement de type constructeur ou méthode.

La syntaxe est la suivante :

@param nom_du_parametre description_du_parametre

Ce tag génère une ligne dans la section Parameters: avec son nom et description dans la documentation. La description peut tenir sur plusieurs lignes.

Exemple extrait de la documentation de l'API du JDK :



Par défaut, ce tag n'est pas pris en compte par javadoc. Pour qu'il soit pris en compte il faut utiliser l'option −author.

Il faut utiliser un tag @param pour chaque paramètre en respectant l'ordre des paramètres dans la signature.

39.3.5. Le tag @return

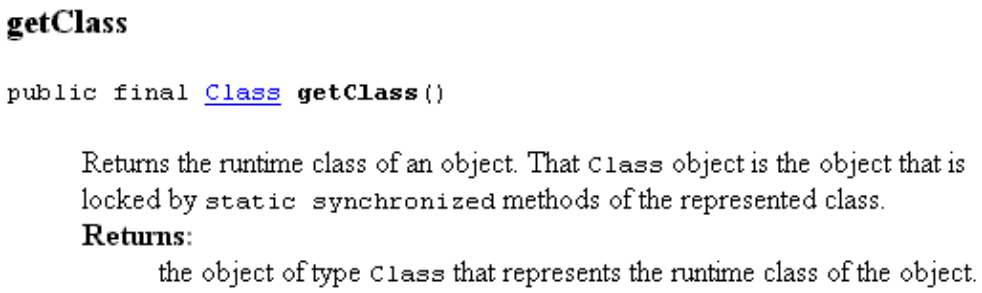
Ce tag permet de préciser la valeur de retour. Ce tag doit être utilisé uniquement pour un élement de type méthode qui renvoie une valeur.

La syntaxe est la suivante :

@return description_retour

Ce tag génère une ligne dans la section Returns: avec sa description dans la documentation. La description peut tenir sur plusieurs lignes.

Exemple extrait de la documentation de l'API du JDK :



39.3_6. La tag @see

Ce tag permet de définir des liens vers d'autres éléments de l'API.

```
public static String valueOf(Object obj)
```

Returns the string representation of the `Object` argument.

Parameters:

`obj` - an `Object`.

Returns:

if the argument is `null`, then a string equal to `"null"`; otherwise, the value of `obj.toString()` is returned.

See Also:

`Object.toString()`

39.3.7. Le tag @since

Ce tag permet de preciser depuis quelle version l'élément est utilisable. Ce tag peut être utilisé avec tous les élements.

La syntaxe est la suivante :

@since numero_de_version

Ce tag génère une ligne dans la section Since: avec son numéro de version.

Exemple extrait de la documentation de l'API du JDK :

```
public byte[] getBytes()
```

Convert this `String` into bytes according to the platform's default character encoding, storing the result into a new byte array.

Returns:

the resultant byte array.

Since:

JDK1.1

39.3.8. Le tag @throws

Ce tag est équivalent au tag @exception

39.3.9. Le tag @version

Ce tag permet de préciser la version d'un élément. Ce tag doit être utilisé uniquement pour un élement de type classe ou interface.

La syntaxe est la suivante :

@version description_de_la_version

Ce tag génère une entrée Version: avec la description de la version dans la documentation. Par défaut, ce tag n'est pas pris en compte par javadoc. Pour qu'il soit pris en compte il faut utiliser l'option −version.

39.4. Les fichiers pour enrichir la documentation des packages

Javadoc permet de fournir un moyen de documenter les packages car ceux ci ne disposent de code source particulier : définir des fichiers dont le nom est particulier.

Ces fichiers doivent être placé dans le répertoire désigné par le package.

Le fichier package.html contient une description du package au format HTML. En plus, il est possible d'utiliser les tags @deprecated, @link, @see et @since.

Le fichier overview.html permet de fournir un résumé de plusieurs packages au format html. Ce fichier doit être placé dans le répertoire qui inclus les packages décrits.

Ce chapitre est en cours d'écriture

40. Java et UML

Ce chapitre est en cours d'écriture

40.1. Présentation de UML

UML qui est l'acronyme d'Unified Modeling Language est aujourd'hui indissociable de la conception objet. UML est le résultat de la fusion de plusieurs méthodes de conception objet des pères d'UML étaient les auteurs : Jim Rumbaugh (OMT), Grady Booch (Booch method) et Ivar Jacobson (use case).

UML a adopté et normalisé par l'OMG (Object Management Group) en 1997.

D'une façon général, UML est une représentation standardisée d'un système orienté objet.

UML n'est pas une méthode de conception mais notation graphique normalisée de présentation de certains concept pour modéliser des systèmes objets. En particulier, UML ne précise pas dans quel ordre et comment concevoir les différents diagrammes qu'il défini. Cependant, UML est indépendant de toute méthode de conception et peut être utilisé avec n'importe quel de ces processus.

Un standard de présentation des concepts permet de faciliter le dialogue entre les différents autres acteurs du projet : les autres analystes, les développeurs, et même les utilisateurs.

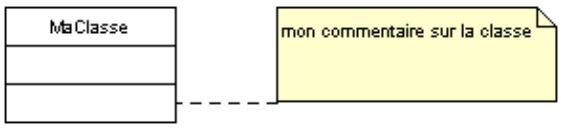
UML est composé de neuf diagrammes :

- des cas d'utilisation
- de séquence
- de collaboration
- d'états−transitions
- d'activité
- de classes
- d'objets
- de composants
- de deploiement

40.2. Les commentaires

Utilisable dans chaque diagramme, UML propose une notation particulière pour indiquer des commentaires.

Exemple :



40.3. Les cas d'utilisation (uses cases)

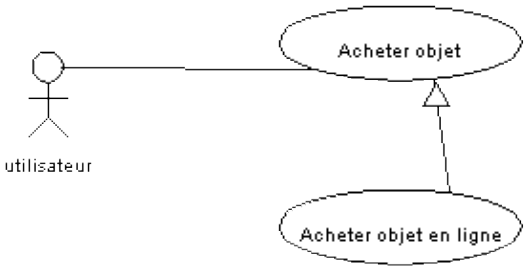
Ils sont développés par Ivar Jacobson et permettent de modéliser des processus métiers en les découpant en cas d'utilisation.

Ce diagramme permet de représenter les fonctionnalités d'un système.Il se compose :

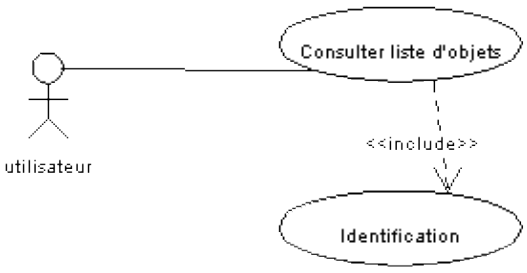
- d'acteurs : ce sont des entités qui utilisent le système à représenter
- les cas d'utilisation : ce sont des fonctionnalités proposées par le système

Un acteur n'est pas une personne désignée : c'est une entité qui joue un rôle dans le système. Il existe plusieurs types de relation qui associe un acteur et un cas d'utilisation :

- la généralisation : cette relation peut être vue comme une relation d'héritage. Un cas d'utilisation enrichie un autre cas en le spécialisant



- l'extension (stéréotype <>) : le cas d'utilisation complète un autre cas d'utilisation
- l'inclusion (stéréotype <>) : le cas d'utilisation utilise un autre cas d'utilisation



Les cas d'utilisation sont particulièrement intéressant pour recencer les différents acteurs et les différentes fonctionnalités d'un système.

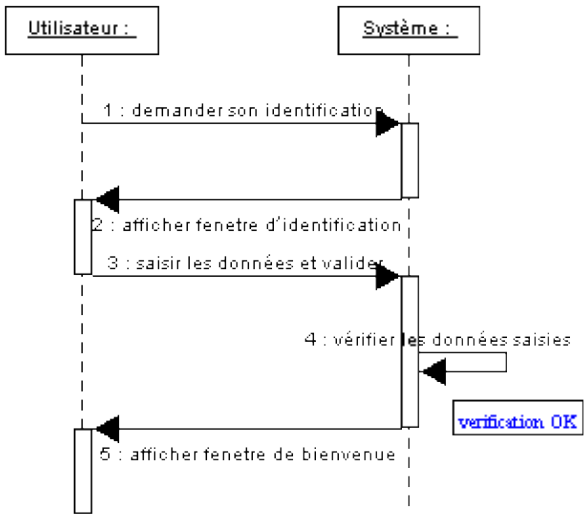
La simplicité de ce diagramme lui permet d'être rapidement compris par des utilisateurs non informaticien. Il est d'ailleur très important de faire participer les utilisateurs tout au long de son évolution.

Le cas d'utilisation est ensuite détaillée en un ou plusieurs scénarios. Un scénario est une suite d'échanges entre des acteurs et le système pour décrire un cas d'utilisation dans un contexte particulier. C'est est un enchainement précis et ordonnés d'opérations pour réaliser le cas d'utilisation.

Si le scénario est trop " volumineux ", il peut être judicieux de découper le cas d'utilisation en plusieurs cas d'utilisation et d'utiliser les relations appropriées.

Un scénario peut être représenter par un diagramme de séquence ou sous une forme textuelle. La première forme est très visuelle

Exemple :



La seconde facilite la representation des opérations alternative.

Les cas d'utilisation permettent de modéliser des concepts fonctionnels. Il ne précise pas comment chaque opération sera implémenter techniquement. il faut rester le plus abstrait possible dans les concepts qui s'approche de la partie technique.

Le découpage d'un système en cas d'utilisation n'est pas facile car il faut trouver un juste mileu entre un découpage faible (les scénarios sont importants) et un découpage faible (les cas d'utilisation se réduise à une seule opération.

40.4. Le diagramme de séquence

Il permet de modéliser les échanges de message entre les différents objets dans le contexte d'un scénario précis

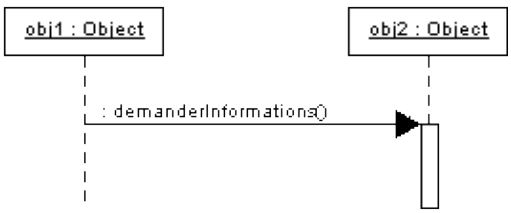
Il permet de répresenter les intéractions entre differentes entités. Il s'utilise essentiellement pour décrire les scénarios d'un cas d'utilisation (les entités sont les acteurs et le système) ou décrirer des échanges entre objets.

Dans le premier cas, les intéractions sont des actions qui sont réalisée par une entité.

Dans le sécond cas, les intérations sont des appels de méthode

Les intérations peuvent être de deux types

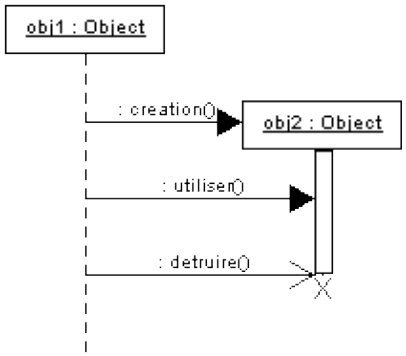
- synchrone : l'émetteur attend une réponse du récepteur



- asynchrone : l'emetteur poursuit son execution sans attendre de réponse

Un diagramme de séquence peut aussi représenter le cycle de vie d'un objet.

Exemple :



40.5. Le diagramme de collaboration

Il permet de modéliser la collaboration entre les différents objets.

40.6. Le diagramme d'états−transitions

Il permet de réprésenter les différents états d’un objet à état fini.

40.7. Le diagramme d'activités

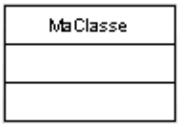
40.8. Le diagramme de classes

Ce shéma représente les différentes classes : il détaille le contenu de chaque classe mais aussi les relations qui peuvent exister entre les différentes classes.

Une classe est représentée par un rectangle séparée en trois parties :

- la première partie contient le nom de la classe
- la seconde contient les attributs de la classe
- la dernière contient les méthodes de la classe

Exemple :

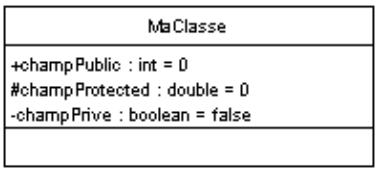


Exemple :

```
public class MaClasse {
}
```

Les attributs d'une classe

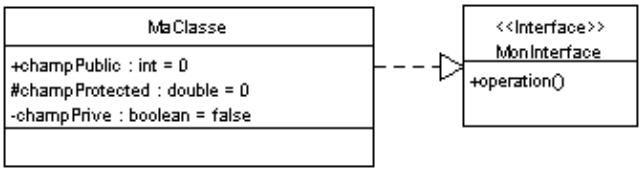
Exemple :



Exemple :

```
public class MaClasse {

    public int champPublic = 0;
    protected double champProtected = 0;
   private boolean champPrive = false;

}
```

Implementation d'une interface



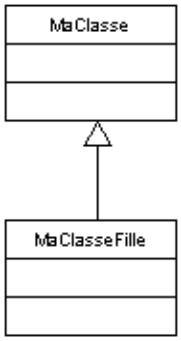
Exemple :

```
public class MaClasse implements MonInterface {
    public int champPublic = 0;
    protected double champProtected = 0;
    private boolean champPrive = false;
```

```
    public operation() {

    }
}
```

La relation d’héritage

Exemple :



Exemple :

```
public class MaClasseFille extends MaClasse {

}
```

40.9. Le diagramme d'objets

40.10. Le diagramme de composants

40.11. Le diagramme de déploiement

41. Des normes de développement

41.1. Introduction

Le but de ce chapitre est de proposer un ensemble de conventions et de règles pour faciliter la compréhension et donc la maintenance du code.

Ces règles ne sont pas à suivre explicitement à la lettre : elles sont uniquement présentées pour inciter le ou les développeurs à définir et à utiliser des règles dans la réalisation de son code surtout dans le cadre d'un travail en équipe. Les règles proposées sont celles couramment utilisées. Il n'existe cependant pas de règles absolues et chacun pourra utiliser tout ou partie des règles proposées.

La définition de conventions et de règles est importante pour plusieurs raisons :

- La majorité du temps passé à coder est consacré à la maintenante évolutive et corrective d'une application
- Ce n'est pas toujours voir rarement l'auteur du code qui effectue ces maintenance
- ces règles facilitent la lisibilité et donc la compréhension du code

Le contenu de ce document est largement inspiré par les conventions de codage proposées par Sun à l'URL suivante : http://java.sun.com/docs/codeconv/index.html

41.2. Les fichiers

Java utilise des fichiers pour stocker les sources et le byte code des classes.

41.2.1. Les packages

Les packages permettent de grouper les classes sous une forme hiérarchisée. Le choix des critères de regroupement est laissé aux développeurs.

Il est préférable de regrouper les classes par packages selon des critères fonctionnels.

Les fichiers inclus dans un package doivent être insérer dans une arborescence de répertoires équivalentes.

41.2.2. Le nom de fichiers

Chaque fichier source ne doit contenir qu'une seule classe ou interface publique. Le nom du fichier doit être identique au nom de cette classe ou interface publique en respectant la casse.

Il faut éviter dans ce nom d'utiliser des caractères accentués qui ne sont pas toujours utilisable par tous les systèmes d'exploitation.

Les fichiers sources ont pour extension .java car le compilateur javac fourni avec le J.D.K. utilise cette extension

Exemple :

```
javac MaClasse.java
```

Les fichiers binaires contenant le byte−code ont pour extension .class car le compilateur génère un fichier avec cette extension à partir du fichier source .java correpondant. De plus, elle est obligatoire pour l'interpréteur java qui l'ajoute automatiquement au nom du fichier fourni en paramètre.

Exemple :

```
java MaClasse
```

41.2.3. Le contenu des fichier sources

Un fichier ne devrait pas contenir plus de 2 000 lignes de code.

Des interfaces ou classes privées ayant une relation avec la classe publique peuvent être rassemblées dans un même fichier. Dans ce cas, la classe publique doit être la première dans le fichier.

Chaque fichier source devrait contenir dans l'ordre

1. un commentaire concernant le fichier
2. les clauses concernant la gestion des packages (la déclaration et les importations)
3. les déclarations de classes ou de l'interface

41.2.4. Les commentaires de début de fichier

Chaque fichier source devrait commencer par un commentaire multi−lignes contenant au minimum des informations sur le nom de la classe, la version, la date, éventuellement le copyright et tout autres commentaires utiles :

Exemple :

```
/*
 * Nom de classe : MaClasse
 *
 * Description   : description de la classe et de soin rôle
 *
 * Version       : 1.0
 *
 * Date          : 23/02/2001
 *
 * Copyright     : moi
 */
```

41.2.5. Les clauses concernant les packages.

La première ligne de code du fichier devrait être une clause package indiquant à quel paquetage appartient la classe. Le fichier source doit obligatoirement être inclus dans une arborescence correspondante au nom du package.

Il faut indiquer ensuite l'ensemble des paquetages à importer : ceux dont les classes vont être utilisées dans le code.

Exemple :

```
package monpackage;
```

```
import java.util.*;
import java.text.*;
```

41.2.6. La déclaration des classes et des interfaces

Les différents éléments qui composent la définition de la classe ou de l'interface devraient être indiqués dans l'ordre suivant :

1. les commentaires au format javadoc de la classe ou de l'interface
2. la déclaration de la classe ou de l'interface
3. les variables de classes (déclarées avec le mot clé static) triées par ordre d'accessibilité : d'abord les variables déclarées public, protected, package friendly (sans modificateur d'accès) et enfin private
4. les variables d'instances triées par ordre d'accessibilité : d'abord les variables déclarées public, protected, package friendly (sans modificateur d'accès) et enfin private
5. le ou les constructeurs
6. les méthodes : elles seront regroupées par fonctionnalités plutôt que selon leur accessibilité

41.3. La documentation du code

Il existe deux types de commentaires en java :

- les commentaires de documentation : ils permettent en respectant quelques règles d'utiliser l'outils javadoc fourni avec le J.D.K. qui formate une documentation des classes, indépendante de l'implémentation du code,
- les commentaires de traitements : ils fournissent un complément d'information dans le code lui−même.

Les commentaires ne doivent pas être entourés par de grands cadres dessinés avec des étoiles ou d'autres caractères.

Les commentaires ne devraient pas contenir de caractères spéciaux tel que le saut de page.

41.3.1. Les commentaires de documentation

Les commentaires de documentation utilisent une syntaxe particulière utilisée par l'outil javadoc de Sun pour produire une documentation standardisée des classes et interfaces au format HTML. La documentation de l'API du J.D.K. est le résultat de l'utilisation de cet outils de documentation

41.3.1.1. L'utilisation des commentaires de documentation

Cette documentation concerne les classes, les interfaces, les constructeurs, les méthodes et les champs.

La documentation est définie entre les caractères /** et */ selon le format suivant :

Exemple :

```
/**
 *  Description de la methode
 */
public void maMethode() {
```

La première ligne de commentaires ne doit contenir que /**

Les lignes de commentaires suivantes doivent obligatoirement commencer par un espace et une étoile. Toutes les

premières étoiles doivent être alignées.

La dernière ligne de commentaires ne doit contenir que */ précédé d'un espace.

Un tel commentaire doit être défini pour chaque entité : une classe, une interface et chaque membres (variables et méthodes).

Javadoc définit un certain nombre de tags qu'il est possible d'utiliser pour apporter des précisions sur plusieurs informations.

Ces tags permettent de définir des caractéristiques normalisées. Il est possible d'inclure dans les commentaires des tags HTML de mise en forme (PRE, TT, EM ...) mais il n'est pas recommandé d'utiliser des tags HTML de structure tel que Hn, HR, TABLE ... qui sont utilisés par javadoc pour formater la documentation

Il faut obligatoirement faire précéder l'entité documentée par son commentaire car l'outil associe la documentation à la déclaration de l'entité qui la suit.

41.3.1.2. Les commentaires pour une classe ou une interface

Pour les classes ou interfaces, javadoc définit les tags suivants : @see, @version, @author, @copyright, @security, @date, @revision, @note

Les tags @copyright, @security, @date, @revision et @note ne sont pas traités par javadoc.

Exemple :

```
/**
 * NomClasse – description de la classe
 * explication supplémentaire si nécessaire
 *
 * @version1.0
 *
 * @see UneAutreClasse
 * @author Jean Michel D.
 * @copyright (C) moi 2001
 * @date 01/09/2000
 * @notes notes particulières sur la classe
 *
 * @revision référence
 *           date 15/11/2000
 *           author Michel M.
 *           raison  description
 *           description supplémentaire
 */
```

41.3.1.3. Les commentaires pour une variable de classe ou d'instance

41.3.1.4. Les commentaires pour une méthode

Pour les méthodes, javadoc définit les tags suivants : @see, @param, @return, @exception, @author, @note

Le tag @note n'est pas traité par javadoc.

Exemple :

```
/**
 * nomMethode – description de la méthode
 *              explication supplémentaire si nécessaire
 *
```

```
 *     exemple d'appel de la methode
 * @return      description de la valeur de retour
 * @param       arg1 description du 1er argument
 *    :           :         :
 * @param       argN description du Neme argument
 * @exception   Exception1  description de la première exception
 *    :           :         :
 * @exception ExceptionN  description de la Neme exception
 *
 * @see UneAutreClasse#UneAutreMethode
 * @author   Jean Dupond
 * @date     12/02/2001
 * @note     notes particulières.
 */
```

Remarques :

- @return ne doit pas être utilisé avec les constructeurs et les méthodes sans valeur de retour (void)
- @param ne doit pas être utilisé si il n'y a pas de paramètres
- @exception ne doit pas être utilisé si il n'y pas d'exception propagée par la méthode
- @author doit être omis si il est identique à celui du tag @author de la classe
- @note ne doit pas être utilisé si il n'y a pas de note

41.3.2. Les commentaires de traitements

Ces commentaires doivent ajouter du sens et des précisions au code : ils ne doivent pas reprendre ce que le code exprime mais expliquer clairement son rôle.

Tous les commentaires utiles à une meilleure compréhension du code et non inclus dans les commentaires de documentation seront insérés avec des commentaires de traitements. Il existe plusieurs styles de commentaires :

- les commentaires sur une ligne
- les commentaires sur une portion de ligne
- les commentaires multi−lignes

Il est conseillé de mettre un espace après le délimiteur de début de commentaires et avant le délimiteur de fin de commentaires lorsqu'il y en aun, afin d'améliorer sa lisibilité.

41.3.2.1. Les commentaires sur une ligne

Ces commentaires sont définis entre les caractères /* et */ sur une même ligne

Exemple :

```
if (i < 10) {
   /* commentaires utiles au code */
   ...
}
```

Ce type de commentaires doit être précédé d'une ligne blanche et doit suivre le niveau d'indentation courant.

41.3.2.2. Les commentaires sur une portion de ligne

Ce type de commentaires peut apparaître sur la ligne de code qu'elle commente mais il faut inclure un espace conséquent qui permette de séparer le code et le commentaire.

Exemple :

```
i++;                        /* commentaires utiles au code */
```

Si plusieurs lignes qui se suivent contiennent chacune un tel commentaire, il faut les aligner :

Exemple :

```
i++;                        /* commentaires utiles au code */
j++;                        /* second commentaires utiles au code */
```

41.3.2.3. Les commentaires multi−lignes

Exemple :

```
/*
 * Commentaires utiles au code
 */
```

Ce type de commentaires doit être précédé d'une ligne blanche et doit suivre le niveau d'indentation courant.

41.3.2.4. Les commentaires de fin de ligne

Ce type de commentaire peut délimiter un commentaire sur une ligne complète ou une fin de ligne.

Exemple :

```
i++;         // commentaires utiles au code
```

Ce type de commentaires peut apparaître sur la ligne de code qu'elle commente mais il faut inclure un espace conséquent qui permette de séparer le code et le commentaire.

Si plusieurs lignes qui se suivent contiennent chacune un tel commentaire, il faut les aligner :

Exemple :

```
i++;                        // commentaires utiles au code
j++;                        // second commentaires utiles au code
```

L'usage de cette forme de commentaires est fortement recommandé car il est possible d'inclure celui ci dans un autre de la forme /* */ et ainsi mettre en commentaire un morceau de code incluant déjà des commentaires.

41.4. Les déclarations

41.4.1. La déclaration des variables

Il n'est pas recommandé d'utiliser des caractères accentués dans les identifiants de variables, cela peut éventuellement poser des problèmes dans le cas ou le code est édité sur des systèmes d'exploitation qui ne les gèrent pas correctement.

Il ne doit y avoir qu'une seule déclaration d'entité par ligne.

Exemple :

```
String nom;
String prenom;
```

Cet exemple est préférable à

Exemple :

```
String nom, prenom;          //ce type de déclaration n'est pas recommandée
```

Il faut éviter de déclarer des variables de types différents sur la même ligne même si cela est accepté par le compilateur.

Exemple :

```
int age, notes[];            // ce type de déclaration est à éviter
```

Il est préférable d'aligner le type, l'identifiant de l'objet et les commentaires si plusieurs déclarations se suivent pour retrouver plus facilement les divers éléments.

Exemple :

```
String          nom          //nom de l'eleve
String          prenom       //prenom de l'eleve
int             notes[]      //notes de l'eleve
```

Il est fortement recommandé d'initialiser les variables au moment de leur déclaration.

Il est préférable de rassembler toutes les déclarations d'un bloc au début de ce bloc. (un bloc est un morceau de code entouré par des accolades).

La seule exception concerne la déclaration de la variable utilisée comme index dans une boucle.

Exemple :

```
for (int i = 0 ; i < 9 ; i++) { ... }
```

Il faut proscrire la déclaration d'une variable qui masque une variable définie dans un bloc parent afin de ne pas complexifier inutilement le code.

Exemple :

```
int taille;
...
void maMethode() {
    int taille;
}
```

41.4.2. La déclaration des classes et des méthodes

Il ne doit pas y avoir d'espaces entre le nom d'une méthode et sa parenthèse ouvrante.

L'accolade ouvrante qui définie le début du bloc de code doit être à la fin de la ligne de déclaration.

L'accolade fermante doit être sur une ligne séparée dont le niveau d'indentation correspond à celui de la déclaration.

Une exception tolérée concerne un bloc de code vide : dans ce cas les deux accolades peuvent être sur la même ligne.

La déclaration d'une méthode est précédée d'une ligne blanche.

Exemple :

```
class MaClasse extends MaClasseMere {
   String nom;
   String prenom;
   MaClasse(String nom, String prenom) {
      this.nom = nom;
      this.prenom = prenom;
   }
   void neRienFaire() {}
}
```

Il faut éviter d'écrire des méthodes longues et compliquées : le traitement réalisé par une méthode doit être simple et fonctionnel. Cela permet d'écrire des méthodes réutilisables dans la classe et facilite la maintenance. Cela permet aussi d'éviter la redondance de code.

Java propose deux syntaxes pour déclarer une méthode qui retourne un tableau : la première syntaxe est préférable.

Exemple :

```
public int[] notes() {     // utiliser cette forme
public int notes()[] {
```

Il est fortement recommandé de toujours initialiser les variables locales d'une méthode lors de leur déclaration car contrairement aux variables d'instances, elles ne sont pas implicitement initialisées avec une valeur par défaut selon leur type.

41.4.3. La déclaration des constructeurs

Elle suit les mêmes règles que celles utilisées pour les méthodes.

Il est préférable de définir explicitement le constructeur par défaut (le constructeur sans paramètre). Soit le constructeur par défaut est fourni par le compilateur et dans ce cas il est préférable de le définir soit il existe d'autres constructeurs et dans ce cas le compilateur ne définit pas de constructeur par défaut.

Il est préférable de toujours initialiser les variables d'instance dans un constructeur soit avec les valeurs fournies en paramètres du constructeur soit avec des valeurs par défaut.

Exemple :

```
class Personne {
   String nom;
   String prenom;
   int    age;

   Personne() {
```

```
      this( "Inconnu", "inconnu", -1 );
   }

   Personne( String nom, String prenom, int age ) {
      this.name    = nom;
      this.address = prenom;
      this.age     = age;
   }
}
```

Il est possible d'appeler un constructeur dans un autre constructeur pour faciliter l'écriture.

Il est recommandé de toujours appeler explicitement le constructeur hérité lors de la redéfinition d'un constructeur dans une classe fille grâce à l'utilisation du mot clé super.

Exemple :

```
class Employe extends Personne {
   int matricule;
   Employee() {
      super();
      matricule = -1;
   }

   Employee(String nom, String prenom, int age, int matricule) {
      super(nom, prenom, age);
      this.matricule = matricule;
   }
}
```

Il est conseillé de ne mettre que du code d'initialisation des variables d'instances dans un constructeur et de mettre les traitements dans des méthodes qui seront appelées après la création de l'objet.

41.4.4. Les conventions de nommage des entités

Les conventions de nommage des entités permettent de rendre les programmes plus lisibles et plus faciles à comprendre. Ces conventions permettent notamment de déterminer rapidement quelle entité désigne un identifiant, une classe ou une méthode.

Entités	Règles	Exemple
Les packages	Toujours écrits tout en minuscules (norme java 1.2)	com.entreprise.projet
Les classes, les interfaces et les constructeurs	La première lettre est en majuscule. Si le nom est composé de plusieurs mots, la première lettre de chaque mot doit être en majuscule, ne pas mettre de caractère underscore '_' Le nom d'une classe peut finir par impl pour la distinguer d'une interface qu'elle implémente. Les classes qui définissent des exceptions doivent finir par Exception.	MaClasse MonInterface MaClasse()
Les méthodes	Leur nom devrait contenir un verbe. La première lettre est obligatoirement une minuscule. Si le nom est composé de plusieurs mots, la première lettre de chaque mot doit être en majuscule sans mettre de caractère underscore '_' Les méthodes pour obtenir la valeur d'un champ doivent commencer par get suivi du nom du champ.	public float calculerMontant() {

	Les méthodes pour mettre à jour la valeur d'un champ doivent commencer ser set suivi du nom du champ Les méthodes pour créer des objets (factory) devraient commencer par new ou create Les méthodes de convertion devraient commencer par to suivi par le nom de la classe renvoyée à la suite de la convertion	
Les variables	La première lettre est obligatoirement une minuscule et ne devrait pas être un caractère dollard '$' ou underscore '_' même si ceux ci sont autorisés. Pour les variables d'instances non publiques, certains recommandent de commencer par un underscore pour éviter la confusion avec le nom d'une variable fournie en paramètre d'une méthode tel que le setter. Si le nom est composé de plusieurs mots, la première lettre de chaque mot doit être en majuscule, ne pas mettre de caractère underscore '_'. Les noms de variables composés d'un seul caractère doivent être évités sauf pour des variables provisoires (index d'une boucle). Les noms communs pour ces variables provisoires sont i,j,k,m et n pour les entiers et c,d et e pour les caractères.	String nomPersonne; Date dateDeNaissance; int i;
Les constantes	Toujours en majuscules, chaque mots est séparés par un underscore '_'. Ces variables doivent obligatoirement être initialisées lors de leur déclaration.	static final int VAL_MIN = 0; static final int VAL_MAX = 9;

41.5. Les séparateurs

L'usage des séparateurs tels que les retours à la ligne, les lignes blanches, les espaces, etc ... permet de rendre le code moins « dense » et donc plus lisibles.

41.5.1.L'indentation

L'unité d'indentation est constituée de 4 espaces. Il n'est pas recommandé d'utiliser les tabulations pour l'indentation.

Il est préférable d'éviter les lignes contenant plus de 80 caractères.

41.5.2. Les lignes blanches

Elles permettent de définir des sections dans le code pour effectuer des séparations logiques.

Deux lignes blanches devraient toujours séparer deux sections d'un fichier source et les définitions des classes et des interfaces.

Une ligne blanche devrait toujours être utilisée dans les cas suivants :

- avant la déclaration d'une méthode,
- entre les déclarations des variables locales et la première ligne de code,

- avant un commentaire d'une seule ligne,
- avant chaque section logique dans le code d'une méthode.

41.5.3. Les espaces

Un espace vide devrait toujours être utilisé dans les cas suivants :

- entre un mot clé et une parenthèse.

Exemple :
`while (i < 10)`

- après chaque virgule dans une liste d'argument
- tous les opérateurs binaires doivent avoir un blanc qui les précèdent et qui les suivent

Exemple :
`a = (b + c) * d`

- chaque expression dans une boucle for doit être séparée par un espace

Exemple :
`for (int i; i < 10; i++)`

- les convertions de type explicites (cast) doivent être suivi d'un espace

Exemple :
`i = ((int) (valeur + 10));`

Il ne faut pas mettre d'espace entre un nom de méthode et sa parenthèse ouvrante.

Il ne faut pas non plus mettre de blanc avant les opérateurs unaires tel que les opérateurs d'incrément '++' et de décrément '−−'.

Exemple :
`i++;`

41.5.4. La coupure de lignes

Il arrive parfois qu'une ligne de code soit très longue (supérieure à 80 caractères).

Dans ce cas, il est recommandé de couper cette ligne en une ou plusieurs en respectant quelques règles :

- couper la ligne après une virgule ou avant un opérateur
- aligner le début de la nouvelle ligne au début de l'expression coupée

Exemple :

```
maMethode(parametre1, parametre2, parametre3,
          parametre4, parametre5);
```

41.6. Les traitements

Même si il est possible de mettre plusieurs traitements sur une ligne, chaque ligne ne devrait contenir qu'un seul traitement

Exemple :

```
i = getSize();
i++;
```

41.6.1. Les instructions composées

Elles correspondent à des instructions qui utilisent des blocs de code.

Les instructions incluses dans ce bloc sont encadrées par des accolades et doivent être indentées. L'accolade ouvrante doit se situer à la fin de la ligne qui contient l'instruction composée. L'accolade fermante doit être sur une ligne séparée au même niveau d'indentation que l'instruction composée.

Un bloc de code doit être définit pour chaque traitement même si le traitement ne contient qu'une seule instruction. Cela facilite l'ajout d'instructions et évite des erreurs de programmation.

41.6.2. L'instruction return

Elle ne devrait pas utiliser de parenthèses sauf si celle ci facilite la compréhension

Exemple :

```
return;
return valeur;
return (isHomme() ? 'M' : 'F');
```

41.6.3. L'instruction if

Elle devrait avoir une des formes suivantes :

```
if (condition) {
   traitements;
}

if (condition) {
   traitements;
} else {
```

```
    traitements;
  }

  if (condition) {
     traitements;
  } else if (condition) {
     traitements;
  } else {
     traitements;
  }
```

Même si cette forme est syntaxiquement correcte, il est préférable de ne pas utiliser l'instruction if sans accolades :

Exemple :

```
    if (i = 10) i = 0;  // cette forme ne doit pas être utilisée
```

41.6.4. L'instruction for

Elle devrait avoir la forme suivante :

```
for ( initialisation; condition; mise à jour) {
     traitements;
  }
```

41.6.5. L'instruction while

Elle devrait avoir la forme suivante :

```
while (condition) {
     traitements;
  }
```

Si il n'y a pas de traitements, la forme est la suivante :

while (condition);

41.6.6. L'instruction do−while

Elle devrait avoir la forme suivante :

```
  do {
     traitements;
  } while ( condition);
```

41.6.7 L'instruction switch

Elle devrait avoir la forme suivante :

```
  switch (condition) {
  case ABC:
     traitements;
  case DEF:
     traitements;
  case XYZ:
     traitements;
  default:
     traitements;
```

```
}
```

Il est préférable de terminer les traitements de chaque cas avec une instruction break.

Toutes les instructions switch devrait avoir un cas ‘default’ en fin d’instruction.

Même si elle est redondante, une instruction break devrait être incluse en fin des traitements du cas ‘default’.

41.6.8. Les instructions try−catch

Elle devrait avoir la forme suivante :

```
try {
   traitements;
} catch (Exception1 e1) {
   traitements;
} catch (Exception2 e2) {
   traitements;
} finally {
   traitements;
}
```

41.7. Les règles de programmation

41.7.1. Le respect des règles d’encapsulation

Il ne faut pas déclarer de variables d’instances ou de classes publiques sans raison valable.

Il est préférable de restreindre l’accès à la variable avec un modificateur d’accès protected ou private et de déclarer des méthodes respectant la conventions instaurées dans les javaBeans : getXxx() ou isXxx() pour obtenir la valeur et setXxx() pour mettre à jour la valeur.

La création de méthodes sur des variables private ou protected permet d’assurer une protection lors de l’accès à la variable (déclaration des méthodes d’accès synchronized) et éventuellement un contrôle lors de la mise à jour de la valeur.

41.7.2. Les références aux variables et méthodes de classes.

Il n’est pas recommandé d’utiliser des variables ou des méthodes de classes à partir d’un objet instancié : il ne faut pas utiliser objet.methode() mais classe.methode().

Exemple à ne pas utiliser si afficher() est une méthode de classe :

```
MaClasse maClasse = new MaClasse();
maClasse.afficher();
```

Exemple à utiliser si afficher() est une méthode de classe :

```
MaClasse.afficher();
```

41.7.3. Les constantes

Il est préférable de ne pas utiliser des constantes numériques en dur dans le code mais de déclarer des constantes avec des noms explicites. Une exception concerne les valeur −1, 0 et 1 dans les boucles for.

41.7.4. L'assignement des variables

Il n'est pas recommandé d'assigner la même valeur à plusieurs variables sur la même ligne :

Exemple :

```
i = j = k;                //cette forme n'est pas recommandée
```

Il ne faut pas utiliser l'opérateur d'assignement imbriqué.

Exemple à proscrire :

```
valeur = (i = j + k ) + m;
```

Exemple à utiliser :

```
i = j + k;
valeur = i + m;
```

Il n'est pas recommandé d'utiliser l'opérateur d'assignation = dans une instruction if ou while afin d'éviter toute confusion.

41.7.5. L'usage des parenthèses

Il est préférable d'utiliser les parenthèses lors de l'usage de plusieurs opérateurs pour éviter des problèmes lies à la priorité des opérateurs.

Exemple :

```
if (i == j && m == n)               // à éviter
if ( (i == j) && (m == n ))         // à utiliser
```

41.7.6. La valeur de retour

Il est préférable de minimiser le nombre d'instruction return dans un bloc de code.

Exemple à eviter :

```
if (isValide()){
   return true;
} else {
  return false;
}
```

Exemple à utiliser :

```
return isValide();
```

Exemple à éviter :

```
if (isValide()) {
   return x;
} else return y;
```

Exemple à utiliser :

```
return (isValide() ? x : y)
```

41.7.7. La codification de la condition dans l'opérateur ternaire ? :

Si la condition dans un opérateur ternaire ? : contient un opérateur binaire, cette condition doit être mise entre parenthèses

Exemple :

```
( i >= 0 ) ? i : -i;
```

41.7.8. La déclaration d'un tableau

Java permet de déclarer les tableaux de deux façons :

Exemple :

```
public int[] tableau = new int[10];
public int tableau[] = new int[10];
```

L'usage de la première forme est recommandée.

42. Les motifs de conception (design patterns)

Ce chapitre est en cours d'écriture

42.1. Présentation

Le nombre de développement avec des technologies orientées objets augmentant, l'idée de réutiliser des techniques pour solutionner des problèmes courants à abouti aux recensements d'un certain nombre de modèles connus sous les motifs de conception.

Ces modèles sont définis pour pouvoir être utilisés avec un maximum de langage orienté objet.

Le nombre de ces modèles est en constante augmentation. Le but de ce chapitre n'est pas de tous les recencer mais de présenter les plus utilisés et de fournir un ou des exemples de leur mise en oeuvre avec Java.

Il est habituel de regrouper ces modèles communs dans trois grandes catégories :

- les modèles de création (creational patterns)
- les modèles de structuration (structural patterns)
- les modèles de comportement (behavioral patterns)

Le motif de conception le plus connus est surement le modele MVC (Model View Controller) mis en oeuvre en premier avec SmallTalk.

42.2. Les modèles de création

Dans cette catégorie, il existe 5 modèles principaux :

Nom	Rôle
Fabrique (Factory)	Interface qui laisse des sous classes créer des objets
Fabrique abstraite (abstract Factory)	
Monteur (Builder)	

Prototype (Prototype)	Création d'objet à partir d'un prototype
Singleton (Singleton)	Classe qui ne pourra avoir qu'une seule instance

42.2.1. Fabrique (Factory)

42.2.2. Fabrique abstraite (abstract Factory)

42.2.3. Monteur (Builder)

42.2.4. Prototype (Prototype)

42.2.5. Singleton (Singleton)

Ce modèle permet de définir une classe dont il ne pourra y avoir qu'une seule instance. Le modèle assure aussi l'accès à cet unique instance.

Ce modèle est particulièrement utile pour le développement d'objets de type gestionnaire. En effet ce type d'objet doit être unique car il gère d'autres objets par exemple un gestionnaire de logs.

Pour mettre en oeuvre ce modèle, il faut :

- créer une instance de la classe stockée dans une variable privée
- empecher l'utilisation du ou des constructeurs
- fournir une méthode qui renvoie l'instance stockée dans la variable privée

Exemple :

```
public class MonSingleton {

  /** Singleton. */
  private static MonSingleton monSingleton = new MonSingleton();

  /**
   * Constructeur de la classe MonSingleton.
   */
  private MonSingleton() {
    super();
  }

  /**
   * Renvoie le singleton.
   */
  public static MonSingleton get() {
    return monSingleton;
  }
```

```
  public void afficher() {
    System.out.println("Singleton");
  }
}
```

Pour vérifier que l'usage du constructeur est impossible, il suffit de compiler une classe qui tente d'en faire usage.

Exemple :

```
public class TestSingleton1 {
  public static void main() {
    MonSingleton ms = new MonSingleton();
    ms.afficher();
  }
}
```

Résultat :

```
C:\java>javac TestSingleton1.java

TestSingleton1.java:4:
No constructor matching MonSingleton() found in class Mon
Singleton.

                MonSingleton ms = new MonSingleton();
                                  ^
1 error
```

Le compilateur indique une erreur car le constructeur a été déclaré private pour empecher son appel.

Pour pouvoir utiliser l'instance de la classe, il faut appeler la méthode qui renvoie l'instance unique.

Exemple :

```
public class TestSingleton2 {
            public static void main(String[] args) {
                        MonSingleton ms = MonSingleton.get();
                        ms.afficher();
            }
}
```

Résultat :

```
C:\java>javac TestSingleton2.java

C:\java>java TestSingleton2
Singleton

C:\java>
```

Il faut impérativement déclarer le ou les constructeurs par défaut et explicitement déclarer un constructeur par défaut pour empecher le compilateur de l’ajouter.

L’instanciation de l’unique instance peut être réalisée de façon statique ou réalisée à la demande. Dans ce cas, la méthode qui renvoie l’instance doit vérifier qu’elle existe et dans la cas contraire, créer l’instance, la stockée dans la variable privée et la renvoyer.

Pour accentuer encore l'assurance de l'unicité de l'instance, il peut être utile de déclarer la classe finale pour éviter que celle ci soit héritée. En effet cette possibilité permettrait de créer un nouveau constructeur dans la classe fille ou de rendre celle ci clonable.

42.3. Les modèles de structuration

42.4. Les modèles de comportement

43. Ant

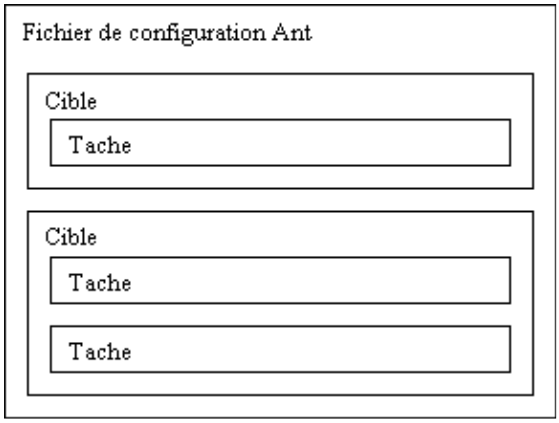
Ce chapitre est en cours d'écriture

Ant est un projet du groupe Apache−Jakarta. Son but est de fournir un outil écrit en java pour permettre la construction d'applications (compilation, exécution de taches post et pré compilation ...). Ces processus de construction d'application sont très importants car ils permettent d'automatiser des opérations répétitives tout au long du cycle de vie de l'application (développement, tests, recettes, mises en production ...). Le site officiel de ant est http://jakarta.apache.org/ant/index.html.

Ant pourrait être comparé au célèbre outils make sous Unix.. Il a été développé pour fournir un outils de construction indépendant de toute plate−forme. Ceci est particulièrement utile pour des projets développés sur et pour plusieurs systèmes ou pour migrer des projets d'un système sur un autre.

Il repose sur un fichier de configuration XML qui décrit les différentes taches qui devront être exécutées par l'outil. Ant fournie un certains nombre de taches courantes qui sont codées sous forme d'objets codés en java. Ces taches sont donc indépendantes du système sur lequel elles seront exécutées. De plus, il est possible d'ajouter ces propres taches en écrivant de nouveaux objets java respectant certaines spécifications.

Le fichier de configuration contient un ensemble de cible (target). Chaque cible contient une ou plusieurs taches. Chaque cible peut avoir une dépendance envers une ou plusieurs autres cibles pour pouvoir être éxécutée.



Les environnements de développement intégrés proposent souvent un outil de construction propriétaire qui son généralement moins souple et moins puissant que Ant. Ainsi des plug−ins ont été développés pour la majorité d'entre eux (JBuilder, Forte, Visual Age ...) pour leur permettre d'utiliser Ant.

Ant possède donc plusieurs atouts : multi plate−forme, configurable grâce à un fichier XML, open source et extensible.

Pour obtenir plus de détails sur l'utilisation de Ant, il est possible de consulter la documentation de la version courante à l'url suivante : http://jakarta.apache.org/ant/manual/index.html

Une version 2 de Ant est en cours de développement.

43.1. Installation

Pour pouvoir utiliser Ant, il faut avoir un JDK 1.1 ou supérieur et installer Ant sur la machine.

43.1.1. Installation sous Windows

Le plus simple est de télécharger la distribution binaire de ant pour Windows : jakarta−ant−version−bin.zip sur le site de Ant.

Il suffit ensuite :

- dézipper le fichier (un répertoire jakarta−ant−version est créer, contenant l'outils et sa documentation)
- ajouter le chemin complet au répertoire bin de Ant à la variable système PATH (pour pouvoir facilement appeler Ant partout dans l'arborescence)
- s'assurer que la variable JAVA_HOME pointe sur le répertoire contenant le JDK
- créer une variable d'environnement ANT_HOME qui pointe sur le répertoire jakarta−ant−version créé lors du dezippe du fichier
- il peut être nécessaire d'ajouter les fichiers .jar contenus dans le répertoire lib de Ant à la variable d'environnement CLASSPATH

Exemple de lignes contenues dans le fichier autoexec.bat :

```
...
set JAVA_HOME=c:\jdk1.3 set
ANT_HOME=c:\java\ant
set PATH=%PATH%;%ANT_HOME%\bin
...
```

43.1.2. Installation sous Linux

43.2. Executer ant

ant s'utilise en ligne de commande avec la syntaxe suivante :

ant [option] [cible]

Par défaut, Ant recherche un fichier nommé build.xml dans le répertoire courant. Ant va alors exécuter la cible par défaut définie dans le projet de ce fichier build.xml.

Il est possible de préciser le nom du fichier de définition en utilisant l'option −buildfile et en la faisant suivre du nom du fichier de configuration.

Exemple :
`ant −buildfile monbuild.xml`

Il est possible de préciser une cible à exécuter. Dans ce cas, Ant exécute les cibles dont dépend la cible précisée et exécute cette dernière.

Exemple : exécuter la cible clean et toutes les cibles dont elle dépend
`ant clean`

43.3. Le fichier build.xml

Le fichier build est un fichier XML qui contient la description du processus de construction de l'application.

Comme tout document XML, le fichier débute par un prologue :

<?xml version="1.0">>

L'élément principal de l'arborescence du document est le projet représenté par le tag qui est donc le tag racine du document.

A l'intérieur du projet, il faut définir les éléments qui le compose :

- les cibles (targets) : ce sont des étapes du projet de construction
- les propriétés (properties) : ce sont des variables qui contiennent des valeurs utilisables par d'autres éléments (cibles ou taches)
- les taches (tasks): ce sont des traitements unitaires à réaliser dans une cible donnée

43.3.1 Le projet

Il est défini par le tag racine dans le fichier build.

Ce tag possède plusieurs attributs :

- name : cet attribut précise le nom du projet
- default : cet attribut précise la cible par défaut à éxécuter si aucune cible n'est précisée lors de l'exécution

- basedir : cet attribut précise le répertoire qui servira de référence pour l'utilisation de localisation relative des autres répertoires.

Exemple :

```
<project name="mon projet" default="compile" basedir=".">
```

43.3.2. Les commentaires :

Les commentaires sont inclus dans un tag <!−− −−>.

Exemple :

```
<!−− Exemple de commentaires −−>
```

43.3.3. Les propriétés

Le tag <property> permet de définir une propriété qui sera utilisée dans le projet : c'est souvent un répertoire ou une variable qui sera utilisée par certaines taches. Leur définition en tant que propriété permet de facilement changer leur valeur une seule fois même si la valeur de la propriété est utilisée plusieurs fois dans le projet.

Exemple :

```
<property name= "nom_appli" value= "monAppli"/>
```

Les propriétés sont immuables et peuvent être définies de deux manières :

- avec le tag <property>
- avec l'option −D sur la ligne de commande lors de l'appel de la commande ant

Pour utiliser une propriété sur la ligne de commande, il faut utiliser l'option −D immediatement suivi du nom de la propriété, suivi du caractère =, suivi de la valeur, le tout sans espace.

Le tag possède plusieurs attributs :

- name : cet attribut défini le nom de la propriété
- value : cet attribut défini la valeur de la propriété
- location : cet attribut permet de définir un fichier avec son chemin absolu. Il peut être utilisé à la place de l'attribut value
- file : cet attribut permet de préciser le nom d'un fichier qui contient la définition d'un ensemble de propriétés. Ce fichier sera lu et les propriétés qu'il contient seront définies.

L'utilisation de l'attribut file est particulièrement utile car il permet de séparer la définition des propriétés du fichier build. Le changement d'un paramètre ne nécessite alors pas de modification dans le fichier xml build.

Exemple :

```
<property file="mesproprietes.properties" />
<property name="repSources" value="src" />
<property name="projet.nom" value="mon_projet" />
<property name="projet.version" value="0.0.10" />
```

L'ordre de définition d'une propriété est très important : Ant gère une priorité sur l'ordre de définition d'une propriété. La règle est la suivante : la première définition d'une propriété est prise en compte, les suivantes sont ignorées.

Ainsi, les propriétés définies via la ligne de commande sont prioritaires par rapport à celles définies dans le fichier build. Il est aussi préférable de mettre le tag <property> contenant un attribut file avant les tag <property> définissant des variables.

Pour utiliser une propriété définie dans le fichier, il faut utiliser la syntaxe suivante :
${nom_propriete}

Exemple :
`${repSources}`

Il existe aussi des propriétés définies par ant et utilisables dans chaque fichier build :

Propriété	Rôle
basedir	chemin absolue du répertoire de travail (cette valeur est précisée dans l'attribut basedir du tag project)
ant.file	chemin absolue du fichier build en cours de traitement
ant.java.version	version de la JVM qui execute ant

43.3.4. Les motifs

Le tag patternset permet de définir un ensemble de motifs pour sélectionner des fichiers.

43.3.5. Les chemins

43.3.6. Les cibles

Le tag définit une cible. Une cible est un ensemble de taches à réaliser dans un ordre précis. Cet ordre correspond à celui des taches décrites dans la cible.

Le tag possède plusieurs attributs :

- name : le nom de la cible. Cet attribut est obligatoire
- description : une brève description de la cible. Cet attribut est optionnel mais il est recommandé de l'utiliser car la plupart des IDE l'affiche lors de l'utilisation de ant
- if : permet de conditionner l'execution par l'existence d'une propriété. Cet attribut est optionnel
- unless : permet de conditionner l'execution par l'inexistance de la définition d'une propriété. Cet attribut est optionnel
- depends : permet de définir la liste des cibles dont dépend la cible. Cet attribut est optionnel

Il est possible de faire dépendre une cible d'un ou plusieurs autres cibles du projet. Lorsqu'une cible doit être éxécutée, Ant s'assure que les cibles dont elle dépend ont été complétement éxécutées préalablement depuis l'éxécution de Ant. Une dépendance est définie grace à l'attribut depends. Plusieurs cibles dépendantes peuvent être listées dans l'attribut depends. Dans ce cas, chaque cible doit être séparée avec une virgule.

43.3.7. Les taches

Une tache est une unité de traitement contenue dans une classe java qui implémente l'interface org.apache.ant.Task. Dans le fichier de configuration, une tache est un tag qui peut avoir des paramètres pour configurer le traitement à réaliser. Une tache est obligatoirement incluse dans une cible.

Ant fourni en standard un certain nombre de taches pour des traitements courants lors du développement en java :

Annexes

Annexe A : GNU Free Documentation License

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111−1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND définiTIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front−matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front−Cover Texts or Back−Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine−readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard−conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine−generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front−Cover Texts on the front cover, and Back−Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine−readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly−accessible computer−network location containing a complete Transparent copy of the Document, free of added material, which the general network−using public has access to download anonymously at no charge using public−standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has

less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front−matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties−−for example, statements of peer review or that the text has been approved by an organization as the authoritative définition of a standard.

You may add a passage of up to five words as a Front−Cover Text, and a passage of up to 25 words as a Back−Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front−Cover Text and one of Back−Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self−contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Annexe B : Glossaire

A

API (Application Programming Interface)	Une API est une bibliothèque qui regroupe des fonctions sous forme de classes pouvant être utilisées pour développer.
Applet	C'est une petite application java compilée, incluse dans une page html, qui est chargée par un navigateur et qui est exécutée sous le contrôle de celui ci. Pour des raisons de sécurité, par défaut, les applets ont des possibilités très resteintes.
AWT (Abstract Window Toolkit)	C'est une bibliothèque qui regroupe des classes pour développer des interfaces graphiques. Ces composants sont dit "lourds" car ils utilisent les composants du système sur lequel ils s'executent. Ainsi, le nombre des composants est volontairement resteints pour ne conserver que les composants présents sur tous les systèmes.

B

BDK (Beans Development Kit)	C'est un outils fourni par Sun qui permet d'assembler des beans de façon graphique pour générer des applications.
Bean	C'est un composant réutilisable. Il possède souvent une interface graphique mais pas obligatoirement.
Byte code	Un programme source java est compilé en byte code. C'est un langage machine indépendant du processeur. Le byte code est ensuite traduit par la machine virtuelle en langage machine compréhensible par le système ou il s'execute. Ceci permet de rendre java indépendant de tout système.

C

CLASSPATH	C'est une variable d'envirronement qui contient les répertoires contenant des bibliothèques utilisables pour la compilation et l'execution du code.
CORBA (Common Object Request Broker Architecture)	C'est un modèle d'objets distribués indépendant du langage de développement des objets dont les spécifications sont fournies par l'OMG.
Core class	C'est une classe standard qui est disponible sur tous les systèmes ou tourne Java.
Core packages	C'est l'ensemble des packages qui composent les API de la plate−forme Java.

D

Deprecated	Terme anglais qui peut être attribué une classe, une interface, un constructeur, une méthode ou un attribut lorque celle ci ne doivent plus être utilisés car Sun ne garantit pas que cet élément sera encore présent dans les prochaine versions de l'API.

E

EJB (Entreprise Java Bean)	Les EJB sont des composants métier qui répondent à des spécifications précises. Il existe deux types d'EJB : EJB Entity qui s'occupe de la persistence des données et EJB session qui gère les traitements. Les EJB doivent s'executer sur un serveur dans un conteneur d'EJB.
Exception	C'est un mécanisme qui permet de gérer les anomalies et les erreurs détectées dans une application en facilitant leur détections et leur traitement. Les exceptions sont largement utilisées et intégrées dans le langage java pour accroitre la sécurité du code.

F

G

Garbage Collector (Ramasse miettes)	C'est un mécanisme intégré à la machine virtuelle qui récupère automatiquement la mémoire inutilisée en restituant les zones de mémoire laissées libres suite à la destruction des objets.

H

HotJava	Navigateur web de Sun écrit en Java
HTML (HyperText Markup Language)	

I

IDL (Interface définition Language)	Langage qui permet de définir des objets devant être utilisé avec CORBA
Interface	C'est une définition de méthodes et de variables de classes que doivent respecter les classes qui l'implémente. Une classe peut implémenter plusieurs interfaces. La classe doit définir toutes les méthodes des interfaces sinon elle est abstraite.

J

J2EE (Java 2 Entreprise Edition)	C'est une version du JDK qui contient la version standard plus un ensemble de plusieurs API permettant le développement d'applications destinées aux entreprises : EJB, Servlet, JSP, JNDI, JMS, JTA, JTS, ...
J2ME (Java 2 Micro Edition)	C'est une version du JDK qui contient le nécessaire pour développer des applications capable de fonctionner dans des environnements limités tels que les assistants personnels (PDA), les téléphones portables ou les systèmes de navigation embarqués
J2SE (Java 2 Standard Edition)	C'est une version du JDK qui contient le nécessaire pour développer des applications et des applets.
JAR (Java ARchive)	Technique qui permet d'archiver avec ou sans compression des classes java et des ressources dans un fichier unique de façon indépendante de toute plate−forme. Ce format supporte aussi la signature électronique.
JDBC (Java Data Base Connectivity)	C'est une API qui permet un accès à des bases de données tout en restant indépendante de celles ci. Un driver spécifique à la base utilisée permet d'assurer cette indépendance car le code java reste le même.
JDC (Java Delevopper Connection)	C'est un service en ligne proposé gratuitement par Sun après enregistrement qui propose de nombreuses ressources sur java (tutorial, cours, information, mailing ...).
JDK (Java Development Kit)	C'est l'environnement de développement Java.Il existe plusieurs versions majeures : 1.0, 1.1, 1.2 (aussi appelée Java 2) et 1.3. Tous les outils fournis sont à utiliser avec une ligne de commandes.
JFC (Java Foundation Class)	C'est un ensemble de classes qui permet de développer des interfaces graphiques plus riches et plus complets qu'avec AWT
JIT Compiler (Just In Time Compiler)	C'est un compiltateur qui compile le byte−code à la volée lors de l'execution des programme pour améliorer les performances.
JMS (Java Messaging Service)	C'est une API qui permet l'échange de messages assynchrones entre applications en utilisant un MOM (Middleware Oriented Message)
JNDI (Java Naming and Directory Interface)	C'est une bibliothèque qui permet un accès aux annuaires de l'entreprise. Plusieurs protocoles sont supportés : LDAP, DNS, NIS et NDS.
JNI (Java Native Interface)	C'est un API qui normalise et permet les appels de code natif dans une application java.
JRE (Java Runtime Environment)	C'est l'environnement d'execution des programmes Java.
JSDK (Java Servlet Development Kit)	C'est un ensemble de deux packages qui permettent le développement des servlets.
JSP (Java Server Page)	C'est une technologie comparable aux ASP de Microsoft mais utilisant Java. C'est une page HTML enrichie de tag JSP et de code Java. Une JSP est traduite en servlet pour être executée. Ceci permet de séparer la logique de présentation et la logique de traitement contenu dans un composant serveur tel que des servlets, des EJB ou des beans.
JTS (Java Transaction Service)	
JVM (Java Virtual Machine)	C'est la machine virtuelle dans laquelle s'éxecute le code java. C'est une application native dépendante du système d'exploitation sur laquelle elle s'execute. Elle répond à des normes dictées par Sun pour

	assurer la portabilité du langage. Il en existe plusieurs développées par plusieurs éditeurs notamment Sun, IBM, Borland, Microsoft, ...

K

L

Layout Manager (gestionnaire de présentation)	Les layout manager sont des classes qui gèrent la disposition des composants d'une interface graphique sans utiliser des coordonnées.

M

N

O

P

Package (Paquetage)	Il permettent des regrouper des classes par critères. Ils impliquent une structuration des classes dans une arborescence correspondant au nom donné au package.

Q

R

Ramasse miette	
RMI (Remote Method Invocation)	C'est une technologie développée par Sun qui permet de faire des appels d'objets distants. Cette technologie est plus facile à mettre en oeuvre que Corba mais elle ne peut appeler que des objets java.

S

Sandbox (bac à sable)	Il désigne un ensemble des fonctionnalités et d'objets qui assure la sécurité des applications. Son composant principale est le gestionnaire de sécurité. Par exemple, Il empeche par défaut à une applet d'accéder aux ressources du système.
Servlet	C'est un composant Java qui s'éxécute côté serveur dans un environnement dédié pour répondre à des requètes. L'usage le plus fréquent est la génération dynamique de page Web. On les compare

	souvent aux applets qui s'éxecutent côté client mais elles n'ont pas d'interface graphique.
Swing	

T

U

V

W

X

Y

Z