Java 5, 6, 7, 8

(avec JDBC)

Table des matières

I - Présentation du langage JAVA.....6
1. Historique et évolution.....6
2. Machine virtuelle JAVA.....7
3. Principales particularités du langage JAVA.....9
4. Structuration des API Java (J2SE / J2EE).....11
5. IDE (Environnement de développement intégré).....12

II - Eléments de base du langage Java.....13
1. Compilation , exécution et IDE.....13
2. Types de données.....15
3. Classes , instances et références.....19
4. Ramasse miettes (G.C.).....26
5. Opérateurs du langage Java.....28
6. Boucles & instructions de Java.....30
7. Chaînes de caractères.....32

8. Tableaux....34
9. Méthode et variables de classes....37

III - Héritage , polymorphisme , interface....40
1. Généralisation / Héritage....40
2. Polymorphisme....44
3. Classes abstraites....46
4. Interfaces....48

IV - Eléments structurants de java....51
1. Packages et archives (.jar)....51
2. Gestion des exceptions....58
3. Présentation des API de log....62
4. Propriétés du système....64
5. Quelques structures de données (java.util)....66
6. Collections (depuis le jdk 1.2)....68
7. Generics (depuis Java 5)....71

V - Classes utilitaires , aspects divers....77
1. Eléments de Java >=5 (jdk 1.5 , 1.6 et 1.7)....77
2. Classes imbriquées (depuis jdk 1.1)....79
3. Quelques classes utilitaires....80
4. Internationalisation....82
5. Mise en forme du texte (java.text)....83
6. Nouveautés apportées par le jdk 1.8....84

VI - Entrées/sorties (io) - fichiers....96
1. Lecture / écriture dans un fichier et à l'écran....96
2. Principe fondamental d'imbrication des flux....96
3. Exemple de code....98
4. (File) répertoires et des attributs sur les fichiers....99
5. Spécificités à connaître et détails intéressants....100
6. Fichier de données accompagnant le code....100
7. Simplification des flux d'entrées - Java 5....100

VII - Introspection et Sérialisation....101
1. Introspection (java.lang.reflect)....101
2. Sérialisation (et persistance élémentaire)....102

VIII - Threads (java)....105
1. Concept de threads....105
2. Gestion des threads avec java....106
3. Synchronisation des threads....108
4. Attente et rendez-vous (wait & notify)....110
5. Autres aspects avancés sur les threads....111

IX - JDBC (accès aux bases de données)....112
1. JDBC : Présentation et structure....112
2. Paramétrage et établissement d'une connexion....113
3. Principaux objets de l'api JDBC....115
4. Lancer un ordre sql....115
5. Effectuer une requête (select)....116
6. Balayer les lignes du résultat....116
7. Accès à la structure de la base (MetaData)....117
8. Gestion des transactions (tout ou rien)....118
9. Préparer et lancer n fois un ordre Sql paramétrable....118
10. Appels de procédures stockées....118
11. Astuce pour fermer proprement les connexions....119
12. Récupérer la valeur d'une clef auto-incrémentée....119
13. Fonctionnalités à partir de la version 2 de JDBC....120
14. Mémento SQL + Mise en oeuvre MySQL....121

X - Api pour IHM/GUI (swing, ...)....123
1. Eléments de base sur AWT/SWING et événements....123
2. Gestion des événements....130

XI - Annexe – Tests unitaires (JUnit 3 et 4)....135
1. Tests unitaires avec JUnit (3 ou 4)....135

XII - Annexe – Strict essentiel MAVEN....139
1. Projet "maven" et dépendances....139
2. Structure & syntaxes (pom.xml)....143

XIII - Annexe – Annotations Java....146
1. Annotations : Présentation et intérêts....146
2. Annotations et méta-annotations prédéfinies....147
3. Création de nouvelles annotations....148

4. Insertion d'annotations au sein d'un code source.....149
5. Analyse et traitement des annotations via l'utilitaire APT (Annotation Processing Tool).....150
6. Accès aux annotations (de rétention RUNTIME) via l'introspection de java 5....153

XIV - Annexe – Java Native Interface (c/c++).....154
1. JNI : Présentation , intérêts et dangers.....154
2. Déclaration et appel d'une méthode native.....154
3. Implémentation d'une méthode native.....154
4. Etablir le lien entre JAVA et le code C:.....155

XV - Annexe – package "java.net" (sockets, http).....156
1. Réseau / URL / Http.....156
2. Récupération du contenu référencé par une URL.....156
3. Contrôle de la connexion liée à une URL.....156
4. Présentation des quelques api "réseaux".....157
5. Dialogue HTTP entre un applet JAVA et un servlet.....157
6. Sockets JAVA (depuis JDK 1.0).....159

XVI - Annexe – Securité "java2" (.policy).....166
1. Sécurité Java2 : Présentation et utilité.....166
2. Sécurité Java2 : Configuration & paramétrages.....166

XVII - Annexe – Structure globale appli.....167
1. Architecture généralement recommandée.....167

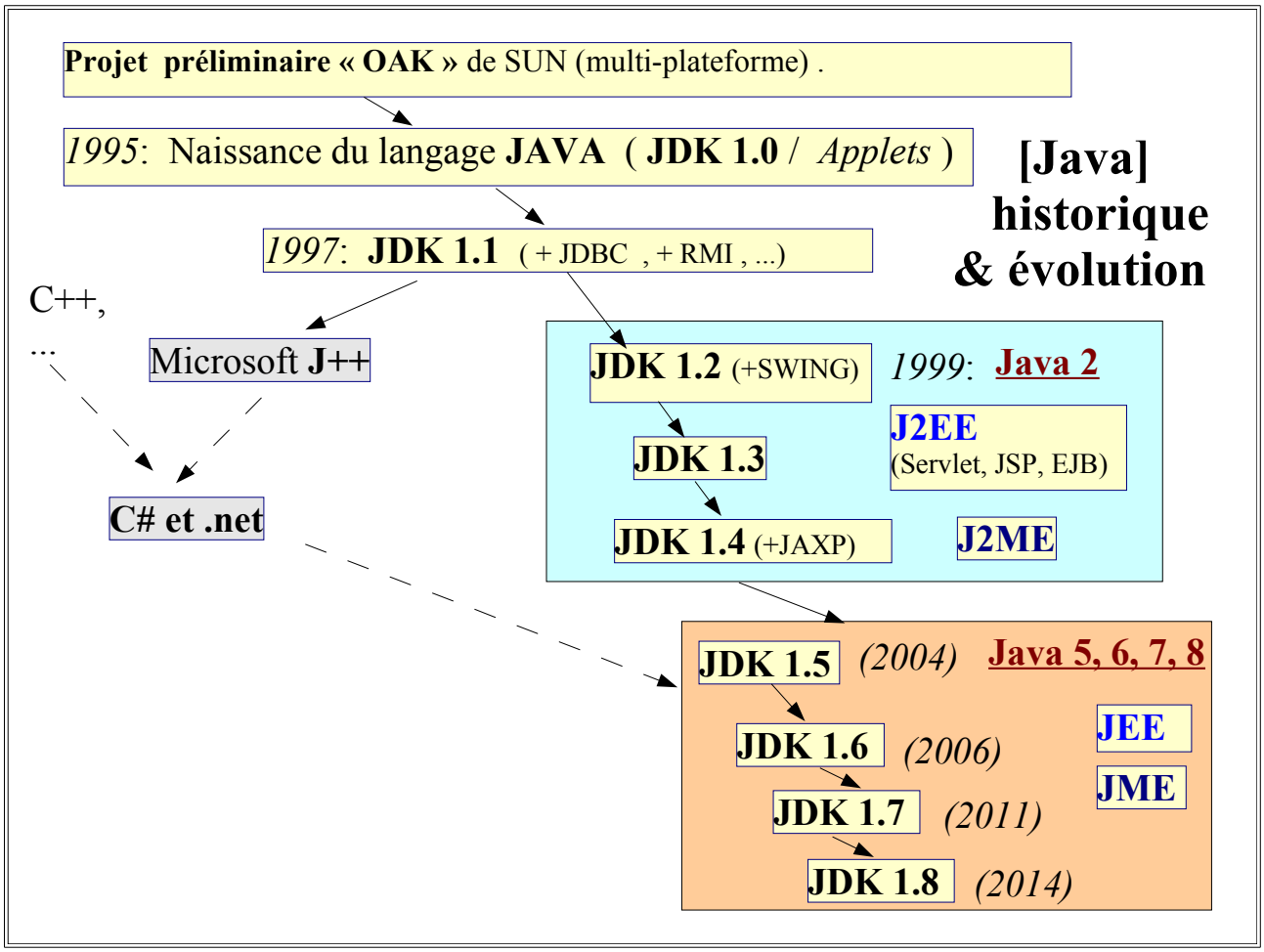
XVIII - Annexe – Prise en main de l'IDE Eclipse.....168
1. Configurations [eclipse , projets].....168
2. IDE et Perspectives.....170

XIX - Annexe – Versions des sources (SVN,GIT).....172
1. Gestionnaires de code source (CVS , SVN, GIT).....172
2. Installation d'un serveur SVN et d'un référentiel SVN.....174
3. Utilisation directe d'un référentiel CVS ou SVN au sein d'un IDE (ex: eclipse). . 176
4. GIT : nouvelle technologie pour contrôler le code source.....177

XX - Annexe – énoncés des TP.....178
1. TP1 (prise en main du jdk).....178
2. TP2 (première classe simple, conventions JavaBean).....178
3. TP3 (classe "AvionV1" avec tableau de "Personne").....179
4. TP4 static – constante , ...179
5. TP5 (classe "Employe" héritant de "Personne").....179
6. TP6 (classe abstraite "ObjetVolant").....180
7. TP7 (interface "Descriptible" ou "Transportable").....180
8. TP8 (Exception):.....180
9. TP9 (Collections & Generics).....181
10. TP10 (Dates & ResourceBundle).....181
11. TP11 (Application ou Applet Dessin en awt/swing):.....181
12. TP12 (Gestion des fichiers) :.....182
13. TP 13 (Accès aux bases de données) :.....182
14. TP 14 (Gestion des threads) :.....182

I - Présentation du langage JAVA

1. Historique et évolution



JDK signifie ***Java Development Kit*** **.**

Les JDK 1.2 et 1.5 ont apportées de grandes nouveautés qui ont modifié le langage en profondeur.

Le terme plate-forme **Java 2** désigne toutes les versions de *Java à partir du JDK 1.2* et englobe également une extension pour les serveurs d'applications : *J2EE* (*Java 2 Enterprise Edition*).

Bien que différents et incompatibles les langages Java et C# comportent beaucoup de points communs (syntaxe et architecture assez proches).

La société **SUN MicroSystem** qui a inventé le langage **JAVA** est le propriétaire officiel du langage et décide de son évolution (en tenant compte des avis de ses partenaires).

L'entreprise "SUN" a été rachetée par "Oracle" .

Oracle est maintenant le **nouveau propriétaire** de "**Java**" .

La version **1.8 du jdk** a apporté quelques grandes nouveautés syntaxiques (**lambda expressions** , streams , …) et propose java-fx à la place de swing .

2. Machine virtuelle JAVA

La principale particularité du langage ***JAVA*** est d'être **multi-plateforme:**

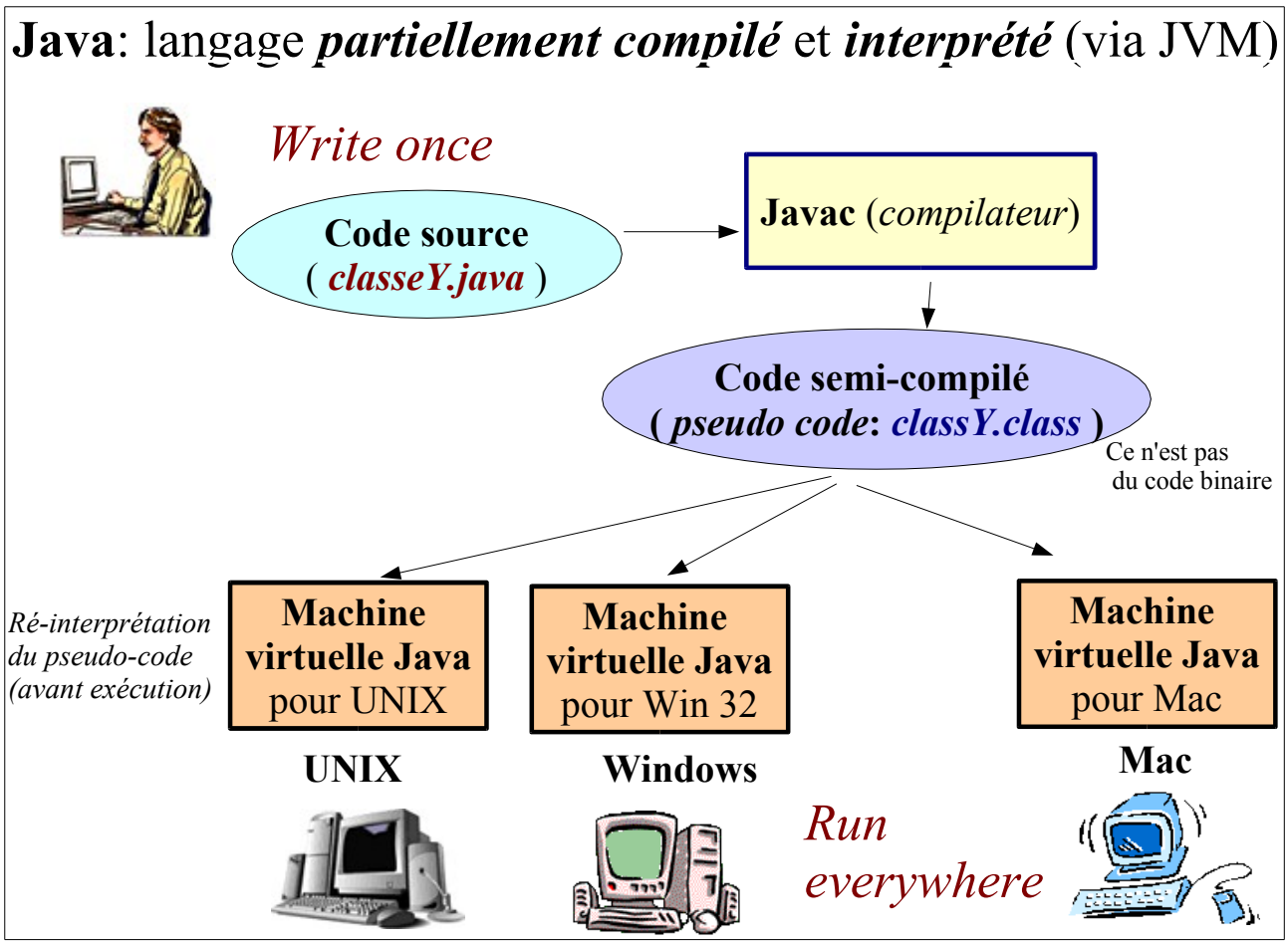
1) Le **développeur écrit du code source** (*classeY.java)* **puis le compile sur n'importe quelle sorte de machine** (Windows , Unix , Linux , ...).

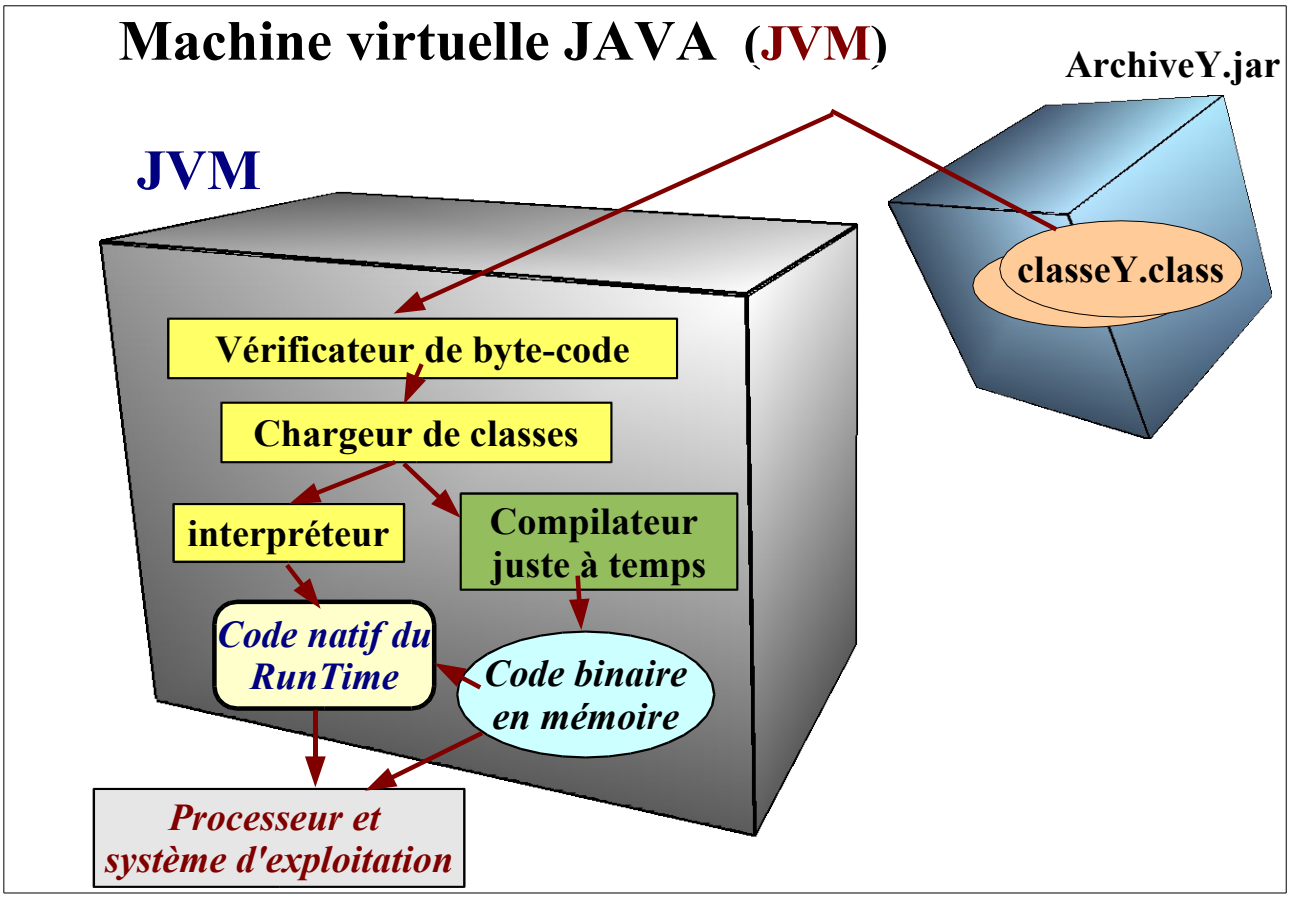
2) Cette **compilation partielle** ne génère pas un code binaire compréhensible que par un certain type de machine mais génère un **pseudo-code portable** appelé "*byte-code*" : *classeY.**class*** .

3) Ce pseudo-code portable est ensuite packagé dans des archives (.jar) puis distribué à travers le réseau vers différentes sortes de machines.

4) N'étant pas dédié à un type précis de machine, **ce pseudo code portable a besoin d'être ré-interprété** par une application spécifique appelée "***machine virtuelle java***" .

A chaque type de plate-forme, correspond une version spécifique de la machine virtuelle java (généralement packagée dans le ***JRE*** = *Java Runtime Environnement*) .





NB:

- Il n'y a pas d'édition de liens à effectuer préalablement (les classes sont chargées au fur et à mesure des besoins du programme lancé au sein de la machine virtuelle) .

- Le **compilateur juste à temps** permet d'**améliorer sensiblement la vitesse d'exécution** d'une application java et est systématiquement activé (sauf option contraire).

Principales contraintes liées aux mécanismes de JAVA:

- L'initialisation de la machine virtuelle, le chargement des classes en mémoire ainsi que la compilation juste à temps sont des opérations assez lourdes qui occasionnent des **temps de démarrage relativement longs**.

- L'ensemble des constituants d'une application Java chargée au sein d'une machine virtuelle **occupe une assez grande place en mémoire vive.**

==>

Ceci explique que ***Java est beaucoup utilisé coté serveur*** (là où une grande consommation mémoire et un temps élevé de démarrage sont moins gênants) .

L'utilisation de java coté client (*ex*: applet , interfaces graphiques Swing, java web start, ...) nécessite des ordinateurs relativement puissants (rapides et bien dotés en mémoire vive).

3. Principales particularités du langage JAVA

Langage générique ***[large palette d'applications]***	Le langage Java peut servir à créer différentes sortes d'entités: ◆ **Applications autonomes** (ne nécessitant qu'une JVM) ◆ **JavaBean** = composants quelconques (pour composer une interface graphique ou pour effectuer des traitements "métier"). ◆ **Servlet** = composant permettant de générer des pages WEB (*Un servlet s'exécute coté serveur*) ◆ **Applet** (mini application s'exécutant coté client , l'affichage s'effectue dans une sous fenêtre du navigateur internet) ◆ ...
Complètement orienté objet	Java est un langage résolument **orienté objet** . Les fonctions globales n'existent pas en java , *toutes les fonctions sont obligatoirement intégrées dans des objets* . Offrant un support à tous les principaux concepts objets (classe, instance, encapsulation, polymorphisme , héritage , ...) , Java permet de **très bien structurer les programmes** (==> **bonne modularité**).
Relativement simple	La **gestion des références** (qui sont plus simples à manipuler que les pointeurs) est **en grande partie automatisée**: *un ramasse miette libère automatiquement les blocs mémoires devenus inutiles*. Bien que ***syntaxiquement proche du C++*** , java ne s'est inspiré que des éléments fondamentaux et simples de ce langage .
Souple, expressif et bien adapté au couches hautes	Souple , modulaire et étant doté de beaucoup d' API prédéfinies , Java est un **langage de prédilection pour les développements "3-tiers" liés à l'informatique de gestion et à internet**.

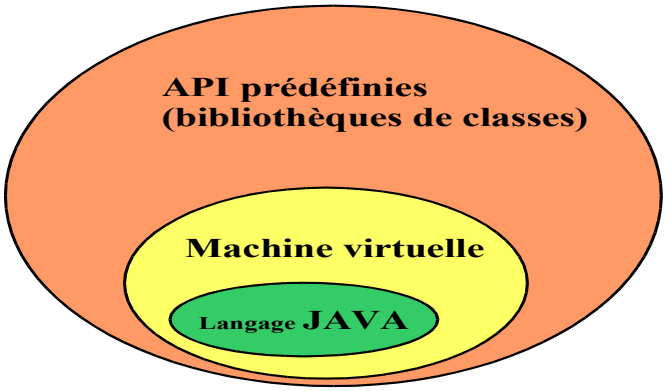
Vitesse d'exécution correcte et interfaçage possible avec le langage C	Par contre, JAVA n'est (pour l'instant) pas du tout approprié pour coder des couches de bas niveau: **Les langages C et C++ restent incontournables pour coder des traitements pointus devant s'exécuter très rapidement sur une plate-forme spécifique (ex: informatique industrielle, calcul scientifique,** ..) Java peut s'interfacer avec du code C ou C++ via **JNI** (*Java Native Interface*) et des DLL locales ou via le protocole SOAP des services WEB.
Robuste et fiable	Langage **fortement typé** ==> beaucoup d'erreurs détectées dès la compilation. **Gestion des exceptions bien structurée** *(try/catch + pile d'appels ==> "Debug" simple et rapide)*.
Introspection	Certains **mécanismes prédéfinis du langage JAVA permettent de récupérer automatiquement une description d'une classe Java** (*liste des attributs et des fonctions internes , types des paramètres , ...*) **même si celle-ci n'est disponible que sous la forme compilée** (*sans code source*) . Ceci constitue un gros point fort (vis à vis du C++) et **permet d'automatiser certains traitements** (persistance des données , proxy dynamique , ...)
Prise en charge (en standard) du multi-threading	==> Java permet d'écrire du **code ré-entrant** que l'on peut écrire de façon **portable** (sans être dépendant d'un système d'exploitation).

4. Structuration des API Java (J2SE / J2EE)

Le langage **JAVA** est associé à une machine virtuelle qui le rend portable sur une large palette de plate-formes (UNIX, Windows , ...) et est **accompagné d'un immense ensemble d'API (bibliothèques de classes) qui sont**:

- **standards** (officialisées par SUN , J2EE , ...)
- **portables** (utilisables sur une multitude de systèmes [Linux , Windows, ...])
- **très souvent gratuites** (Open source ou ...)

Java est donc bien plus qu'un langage informatique , c'est une véritable plate-forme virtuelle:



(plate forme Java) **JavaSE / JavaEE et JavaME**

Java EE
(Java Enterprise Edition)

API supplémentaires (**.jar**) :
pour serveurs d'application
- **Servlet / JSP** (aspects web),
- **JNDI** (accès aux serveurs de noms) ,
- **EJB** (objets métiers en java)

...

Java SE
(Java Standard Edition)

(*Java Development Kit*)
(**JDK** 1.2 , …, 1.6 , 1.7 ou 1.8)
→ **Compilateur java**
+
JRE (*Java Runtime Environnement*)
--> machine virtuelle Java
--> (API standards + code natif)

NB: il existe aussi ***JavaME*** (*Micro Edition*) = **Java**SE ***simplifié/allégé*** pour les **mobiles/smartphones/pda/...** (ex: pour *androïd*).

4.1. Principales API de JAVA

Api	intégration	fonctionnalités
JDBC	API intégré dans **J2SE** mais *Driver JDBC à récupérer ailleurs*	Accès générique aux bases de données relationnelles (==> requêtes SQL)
AWT	**J2SE** *(depuis JDK 1.0)*	Bases du graphisme et du multi-fenêtrage
SWING	**J2SE** *(depuis JDK 1.2)*	Graphisme 2D et contrôles graphiques 100% java
javaFx	***J8SE*** *(depuis JDK 1.8)*	Api graphique plus moderne
SERVLET / JSP	***J2EE***	génération de pages HTML (nécessite un conteneur Web du type Tomcat)
RMI ***(Remote Method Invocation***)	**J2SE** *(depuis JDK 1.1)*	Appels de fonctions à travers le réseau
EJB (**E**nterprise **J**ava **B**ean)	***J2EE***	Objets "métier" en java (nécessite serveur d'application JEE ex: WebSphere_AS, JBoss_AS, ...)
JPA 1 et 2	***J5EE***	**J**ava **P**ersistance **A**pi (ORM / hibernate)
DI , CDI	***J6EE***	Injection de dépendances
JNDI	**J2SE** *(depuis JDK ....)*	Accès à des serveurs de noms (LDAP, ...)
JMS	***J2EE***	Interface java pour MiddleWare orienté message asynchrone.
JAXP (Java Api for Xml Processing)	**J2SE** *(depuis JDK 1.4)*	Parsing XML (SAX & DOM) + transformations XSLT
...		

5. IDE (Environnement de développement intégré)

Le **JDK (JRE + compilateur en mode texte)** ne suffit pas pour programmer de façon confortable.

Un ***environnement de développement graphique*** intégrant au minimum un ***éditeur*** et une ***gestion automatisée des compilations*** permet d'être efficace durant les phases de programmation .

Eclipse *(Open Source)* et **NetBeans** *(de Sun/Oracle)* sont actuellement les deux **IDE Java** qui sont les plus utilisés .

Dans le cadre d'un projet d'entreprise sérieux, **un outil de gestion de version** (**CVS**,**SVN** ou **GIT**) permet de stocker/centraliser le code de tous les développeurs dans un **référentiel partagé en commun** .

En outre le produit "**Maven**" est très souvent utilisé pour **gérer les librairies java (".jar") et leurs inter-dépendances** .

II - Eléments de base du langage Java

1. Compilation , exécution et IDE

1.1. Structuration élémentaire d'un programme java

Un programme Java doit au minimum comporter une classe représentant le point de démarrage de l'application et intégrant la méthode **main()** :

tp/MyApp.java

```
package tp;

public class MyApp {

        public static void main(String args[]) // fonction principale d'une application Java
        {
        System.out.println("Hello World\n");
        }// Fin du main

} // Fin de la classe
```

NB: Le langage Java impose les choses suivantes:

- Le **nom d'une classe** publique doit **obligatoirement correspondre au nom du fichier**.
- Le **nom du package** (contenant la classe) **doit être le même que le nom du répertoire** comportant le fichier lié à la classe.

Conventions importantes:

- Nom de package (répertoire) entièrement en minuscules (*ex*: *tp*)
- **Nom de classe commençant toujours par une majuscule** (*ex*: ***M****yApp*)
- Nom de méthode (fonction) commençant toujours par une minuscules (*ex*: ***p****rintln*())
- Nom de constante entièrement en majuscules (*ex*: Math.***PI***)

Remarque: on peut utiliser un éditeur de texte quelconque (vi , notepad , ...)

1.2. Compilation et exécution

Après s'être placé *(via la commande **cd**)* dans le répertoire contenant le sous répertoire "***tp***" lié au package de l'application précédente,

on lance une **compilation** en invoquant le compilateur **javac** (gratuit et intégré au *JDK / J2SDK*):

```
set PATH=%PATH%;C:\prog\JAVA\j2sdk1.4.2_06\bin
javac  tp\MyApp.java
```

on démarre ensuite l'interprétation de cette application par la machine virtuelle **java** :

```
set CLASSPATH=%CLASSPATH%;.
java   tp.MyApp
```

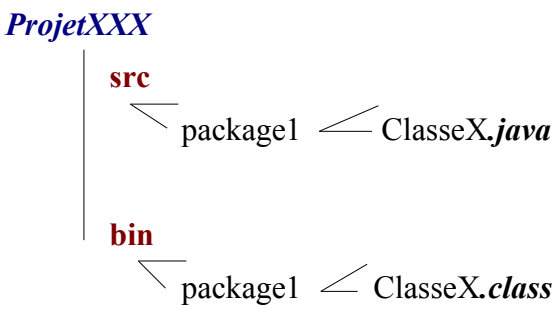
tp.MyApp correspond içi au nom complet de la classe java qui comporte la méthode *main()* .

1.3. Automatisation via les projets des IDE et ANT

En pratique le compilateur ***javac*** du JDK est très rarement directement invoqué depuis une ligne de commande mais est indirectement activé par un **IDE** graphique (tel que **JBuilder** ou **Eclipse**) .

La constitution d'un **projet** comportant différents packages de classes java permettra à l' IDE de lancer automatiquement les compilations nécessaires .

Bien que facultatif , la séparation du code source et du code compilé est souhaitable:



D'autre part , le projet open source "**ANT**" (téléchargeable depuis *www.apache.org*) permet d'écrire en XML des scripts "***build.xml***" (*comparables à des "**Makefile**"*) qui peuvent servir à automatiser des compilations et des lancements d'application JAVA.

2. Types de données

2.1. Vue d'ensemble sur les différents types de données de Java

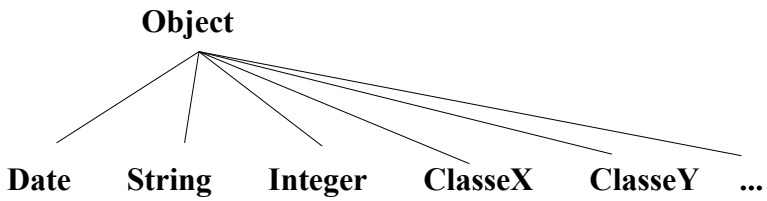
- **types élémentaires (non orientés objet et entièrement orthographiés en minuscule)** :

  int , **short**, **long** , **boolean**, **byte** , **char**, **float** , **double**

- **types "objet" (classes prédéfinies ou pas) :**

  String , **Vector**, **Date** et tous les autres types de données possibles.

Remarque: tous les classes de java dérivent du type générique **Object** :



Un "***Vector***" est une ***Collection*** particulière de références sur des "***Object***" quelconques.

2.2. Types élémentaires (int, double , ...)

Les types de données élémentaires de JAVA sont toujours manipulés par valeur.

Type	*Contenu*	*Valeur défaut*	*Taille (bits)*	*Valeur min.* *Valeur max*
boolean	true or false	false	**1**	
char	caractère Unicode	\u0000	**16**	\u0000 à \uFFFF
byte	entier signé	0	**8**	-128 à 127
short	entier signé	0	**16**	-32 768 à 32 767
int	entier signé	0	**32**	-2 147 483 648 à 2 147 483 647
long	entier signé	0	**64**	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807
float	flottant IEEE 754	0.0	**32**	1.40239846^{E} **-45** à 3.40282347^{E} **+38**
double	flottant IEEE 754	0.0	**64**	4.94…**E-324** à 1.797…**E+308**

Remarque:

Les **types élémentaires** sont directement (ou quasi-directement) mis en relation avec des types de données nativement gérés par le micro processeur . ils occupent peu de place en mémoire (comparés à des types "objet") et sont gérés de façon efficace ==> **bonnes performances**.

NB:

- JAVA ne considère pas que **false** est équivalent à 0 .
- Le mot clef *unsigned* n'existe pas en Java (contrairement au C/C++)
- **'**A**'** ou '**\u**0041' est une valeur littérale de type **char**
- 3.14159**f** est une valeur littérale de type **float**
- 3.14159 ou 3.14159**d** est une valeur littérale de type **double**
- ***0x***a25c est une valeur littérale de type **int** exprimée en **hexadécimal**

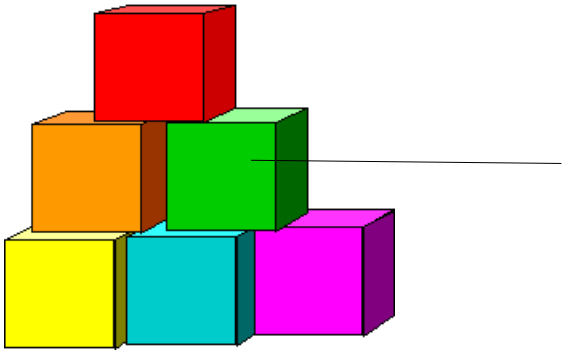
Les types ***float*** et ***double*** peuvent acquérir des valeurs spéciales définies dans *java.lang.Float* et *java.lang.Double*. Ces valeurs spéciales sont des constantes (POSITIVE_INFINITY, NEGATIVE_INFINITY, NaN signifiant *Not A Number* ,…) qui sont généralement le résultat d'une opération en virgule flottante.

L'arithmétique sur les réels en virgules flottantes ne produit jamais d'exception, même dans le cas d'une division par 0.0

Conversions entre types:

```
int i=4 ,j , k;
double x=3.5 , y, z ;
boolean b=true , bb;
String ch;

ch=String.valueOf(i);    j=Integer.parseInt(ch);     System.out.println("j="+j);
ch=String.valueOf(x);    y=Double.parseDouble(ch);   System.out.println("y="+y);
ch=String.valueOf(b);    bb=Boolean.valueOf(ch).booleanValue();

k = (int) x; // i va récupérer la partie entière de x

i=2; j=3; k=i/j; // k=0 [division entre nombres entiers]
z = ((double) i) / ((double) j); // z =0.6666666666
```

2.3. Types "Objets" et notion de référence



Un **objet** est une ***brique de base*** dans un programme.

Un **objet** est une **entité qui rend un certain service** (ex: mémorisation, affichage de données , ...).

Un objet est une entité qui regroupe:

- des **données internes** appelées **attributs** (et parfois propriétés)
- des **traitements (fonctions) internes** appelées **méthodes**

Tous les éléments du langage JAVA qui ne sont pas de types primitifs sont des objets manipulés par référence. C'est le cas des tableaux et des chaînes de caractères.

Une ***référence*** est un ***pointeur caché*** sur un objet créé dynamiquement en mémoire.

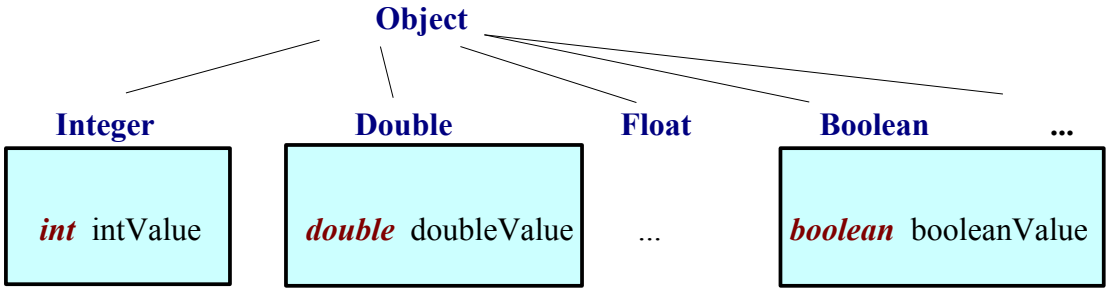
```
ClasseY ref =null;   // déclaration d'une variable ref qui pourra référencer un
                     // futur objet de type ClasseY

ref = new ClasseY(); // création d'un nouvel objet (instance) de type ClasseY .
```

2.4. Wrapper class

Il est quelquefois pratique de pouvoir manipuler un simple nombre entier comme un objet .

Ceci permet par exemple d'insérer une valeur (au départ "non objet") dans une collection d'objets.

A cet effet , le langage Java comporte des classes dites "***Wrapper***" qui correspondent à des **enveloppes "Objet"** permettant d'incorporer des valeurs élémentaires (*de types primitifs*).



Exemple: pour placer un entier nombre de valeur 5 dans une Collection d'objet , il faut créer un objet de la classe **Integer** (*commençant par un **I** majuscule*) qui va lui même incorporer la valeur 5 :

```
java.util.Vector liste = new java.util.Vector();
liste.add (new Integer(5));
```

Inversement pour récupérer la valeur du premier élément de la liste , il faut extraire la valeur primitive de son enveloppe objet via la méthode prédéfinie *intValue*() :

```
Integer  objVal = (Integer) liste.elementAt(0);
int a = objVal.intValue() ;
```

2.5. Boxing / unboxing (depuis Java 5)

Depuis le JDK 1.5 , **les conversions entre les types primitifs (int, double, ...) et les types "Wrapper" (enrobages "objet") correspondants (Integer, Double , ...) sont devenus implicites et automatiques**:

```
java.util.Vector liste = new java.util.Vector();
liste.add (5);    // BOXING (incorporation automatique : new Integer(5))
Integer  objVal = (Integer) liste.elementAt(0);
int a = objVal ;    // UNBOXING (extraction automatique via intValue() )
```

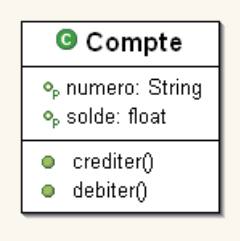
Rappel ==> **Cette grande simplification du code** n'est possible qu'à partir de ***Java 5 (jdk 1.5)***.

3. Classes , instances et références

3.1. Classe d'objet (concepts & syntaxe)

Une **classe** correspond plus ou moins à un **type d'objet**. C'est une **entité qui décrit une structure de données et une liste de méthodes (fonctions internes) opérant sur ces mêmes données**:

Exemple : représentation UML d'une classe Compte:



Code Java associé (fichier Compte.java):

```
package ex;

public class Compte {

   // Attributs (données internes):
        public String numero;
        public float solde;

   // Méthodes (fonctions internes):
        public void debiter(float montant)  { solde = solde - montant; }
        public void crediter(float montant) { solde = solde + montant; }
}
```

3.2. Instances manipulées via des références

Pour utiliser une classe, il faut (à l'extérieur de la définition de celle-ci) :

- Déclarer une variable de type "référence sur un objet de la classe considérée"
- créer un objet ou une instance (exemplaire) de la classe.

```
Compte c1 = null;  // c1 est içi une référence sur un objet de type Compte
...                // qui n'existe pas encore.
c1 = new Compte(); // c1 référence maintenant un nouvel exemplaire de la classe Compte.
```

Remarque importante:

En JAVA, **tous les objets (instances d'une classe)** sont toujours **manipulés par référence** (pointeur caché) et **alloués dynamiquement en mémoire** (directement via le mot clef **new** ou indirectement via d'autres mécanismes).

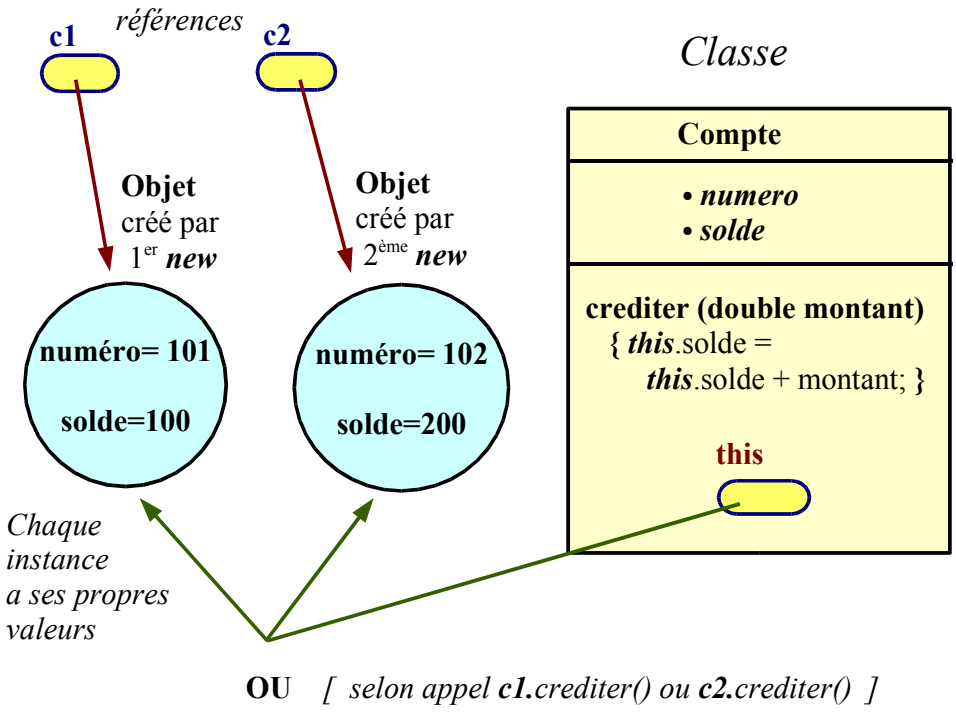
Accès aux attributs et méthodes publiques:

```
public class MyApp {

 public static void main(String[] args) {
        Compte c1, c2;
        c1=new Compte();
        c1.numero = "101";   c1.solde = 100;
        c2 = new Compte();
        c2.numero = "102";   c2.solde = 200;
        c1.crediter(10);   c2.debiter(10);
        System.out.println("Le compte num = " + c1.numero + " a un solde de " + c1.solde );
        System.out.println("Le compte num = " + c2.numero + " a un solde de " + c2.solde );
   }
}
```

NB:

L'écriture ***c1.crediter(10)*** revient à appeler la méthode "*crediter*" depuis l'objet référencé par c1. La méthode en question est alors invoquée avec une référence interne implicite dénommée **this**.
this référence systématiquement l'objet (courant) à partir duquel la méthode à été appelée.

Le code interne de la méthode *crediter* est implicitement converti en
*{ **this**.solde = **this**.solde + montant ; }*



OU *[selon appel **c1.**crediter() ou **c2.**crediter()]*

Au moment où ***c1.crediter(10)*** est appelé la référence spéciale **this** est automatiquement initialisée en y recopiant la valeur (adresse mémoire) de c1. Donc this et c1 pointent vers le même objet. Lorsque le code de la méthode (fonction) *créditer* s'exécute , il manipule les valeurs internes (numéro , solde) de l'objet couramment référencé par this.

Repère Syntaxique:

Appel classique de fonction en langage "C" (non orienté objet):

res = fonctionDeTraitement (structureDeDonnées);

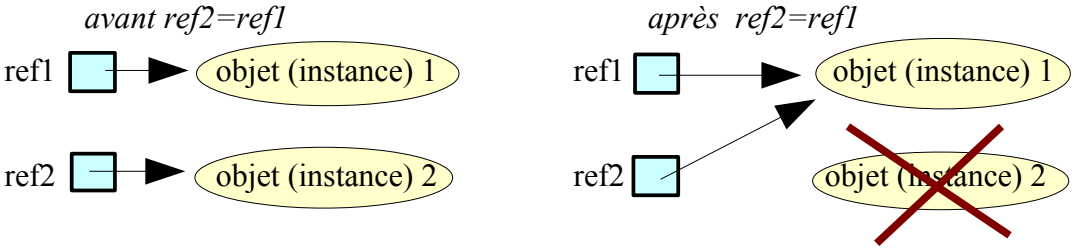
Appel classique de méthode (fonction interne) sur un objet Java:

res = ObjetAvecSesPropresDonnées.méthodeDeTraitement();

3.3. Copies de références et tests d'égalités

Soient 2 références ***ref1*** et ***ref2*** pointant initialement sur 2 objets différents (*instances construites via new*). Alors l'affection ***ref2=ref1*** va faire en sorte que:

- les 2 références ref1 et ref2 vont pointées sur le même objet
- l'ancien objet 2 (jusqu'à référencé par ref2) ne sera plus référencé et sera automatiquement détruit par le ramasse-miettes.




```
if( ref1 == ref2 )
//teste simplement le fait que  ref1 et ref2 référencent bien le même objet.
```

Pour tester si deux objets distincts ont des valeurs internes identiques, il faut utiliser la méthode spéciale **equals** (déclarée au niveau de **Object** et recodée dans les sous classes) :

```
if( obj1.equals(obj2) ) ...
if( chaine1.equals("abc") )...
```

La méthode *clone()* lorsqu'elle existe, permet de déclencher une copie en profondeur:

```
ref3 = ref.clone();  // ref3 référence une nouvelle instance , résultant du clonage
```

3.4. Constructeurs

Lorsqu'un nouvel objet est créé, ses valeurs internes peuvent alors être automatiquement initialisées au moyen d'une fonction particulière appelée ***constructeur***.

Le **constructeur** d'une classe est une **méthode très spéciale** qui doit absolument porté le **même nom de la classe** et qui n'a **pas de type de retour** (*pas de void*).

Exemple:

```
public class Compte  {
...
public Compte  (String   num, double solde_initial )
        {   this.numero =num; this.solde = solde_initial ;  }
...}
```

Utilisation du constructeur:

```
Compte  c = new Compte("103",150.0); // nouveau compte de numéro "103" et de solde 150.0
```

NB: Une classe peut comporter **plusieurs versions** de son constructeur (**fonction surchargée**):

```
public Compte  (String num, double s) { this.numero = num; this.solde = s; }
public Compte() { this.numero = "0"; this.solde = 0; }
public Compte( Compte  c) { this.numero = c.numero;  this.solde = c.solde; }
```

La version à utiliser sera choisie en fonction des arguments présents sur la ligne de code effectuant l'appel:

```
c1 = new Compte();
c2 = new Compte("104",120);                    c3 = new Compte(c2);
```

NB: **Un constructeur peut en interne s'appuyer sur une des autres versions pour développer son code**. On utilise pour cela une ***syntaxe*** faisant intervenir le mot clé ***this*** avec des parenthèses**:**

```
public Compte  (String num, double s) { this.numero = num; this.solde = s; }
public Compte() { this("0",0.0); }
public Compte( Compte c) { this(c.numero, c.solde) ; }
```

Remarque importante:

- Si aucun constructeur n'a été programmé au niveau d'une classe , le langage Java en fabrique un automatiquement sans argument: *refObjX = new ClasseX();*
- Si les seuls constructeurs qui ont été programmés au niveau d'une certaine classe ont tous au moins un paramètre alors on pourra créer une nouvelle instance via *refObjX = new ClasseX(valParam1, ...);* mais on ne pourra plus écrire *refObjX = new ClasseX();*

  ==> **Il est donc très fortement conseillé de programmer** soi même (avant de l'oublier) **le constructeur par défaut (sans argument)**.

3.5. Accesseurs (get/set)

La plupart des classes ***java*** doivent être programmées dans les règles de l'art :

- les attributs (données internes) doivent normalement être déclarés privés (via le mot clef "**private**"). ils ne pourront alors pas être directement modifié depuis une méthode d'une autre classe . On parle de **protection des données**.
- Des ***méthodes publiques*** **get*Xxx*()** et **set*Xxx*(...)** permettant respectivement de ***récupérer*** et de ***modifier indirectement*** la valeur de l'attribut ***xxx***.
- Un constructeur par défaut (sans argument) doit être explicité

Remarque importante:

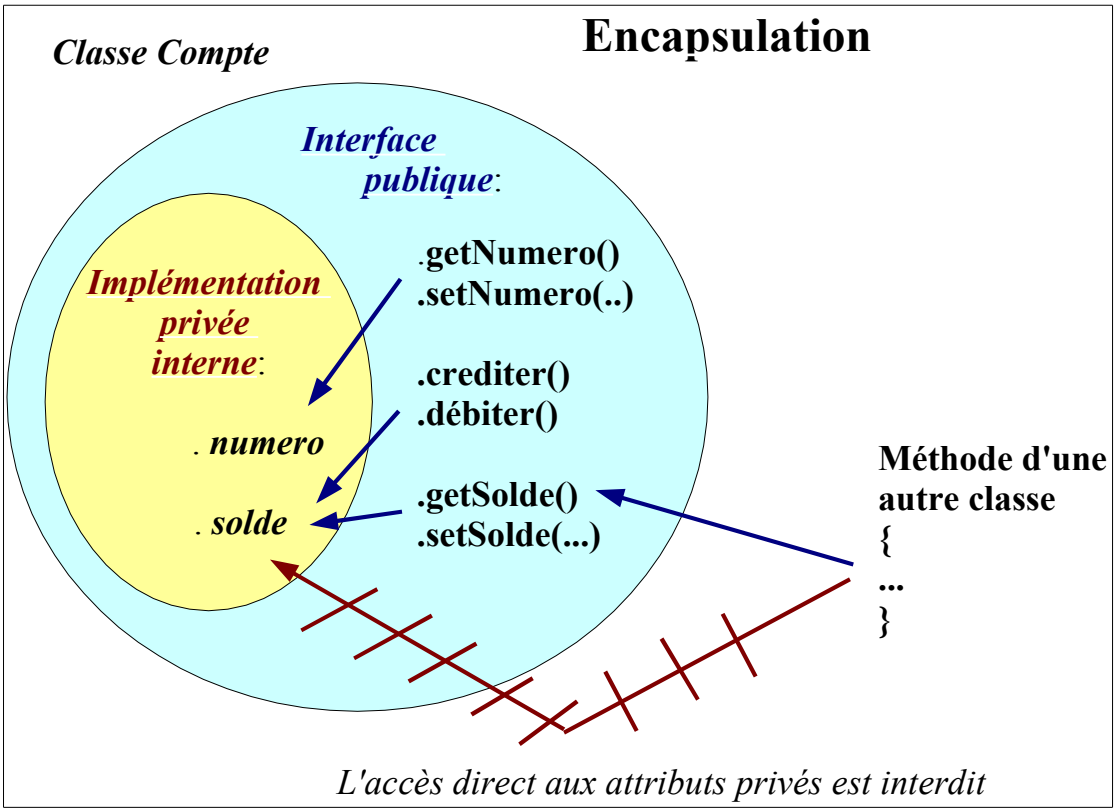
Toute classe java qui respecte les règles précédemment évoquées constitue un "**Java Bean**".

Un **JavaBean** est un composant de base qui peut être facilement réutilisé même si l'on ne dispose pas de son code source.

Nouvelle version de la classe Compte (*respectant les conventions d'un JavaBean*):

```
public class Compte {

// Attributs privés:
        private String numero;
        private float  solde;

// Accesseurs:
        public String getNumero() { return numero;}
        public void setNumero(String numero) { this.numero = numero; }

        public float getSolde() { return solde; }
        public void setSolde(float solde) { this.solde = solde; }

// Méthodes :
        public Compte() { numero="0"; solde=0; }  // constructeur par défaut

        public void debiter(float montant) { solde = solde - montant; }
        public void crediter(float montant) { solde = solde + montant; }
}
```

```
...
c1.setNumero("102");  c1.setSolde(200);
System.out.println("Le compte num = " + c1.getNumero() + " a un solde de " + c1.getSolde() );
```



Principaux intérêts de l'encapsulation :

- Les méthodes de la classe courante sont les seules à pouvoir manipuler les attributs privés internes. La cohérence des données est ainsi plus simple à assurer. Du code externe (issu d'une autre classe) sera obligé de passer par les méthodes **get*Xxx***() et **set*Xxx***() pour récupérer et modifier indirectement les valeurs --> un contrôle est alors possible:

  void setAge(int nouvel_age)
  *{ **if**(nouvel_age >= 0 && nouvel_age < 150) this.age = nouvel_age; }*

- Ceci permet d'effectuer des **éventuelles évolutions** (version 2, ...) au niveau de la représentation interne des données **sans remettre en cause toutes les utilisations externes** d'une certaine classe. C'est un des points clefs de la **modularité**.

- D'autre part, un **objet** ne doit normalement pas être vu comme une simple structure de donnée (que l'on manipule directement de l'extérieur) mais comme une **entité relativement autonome qui sait se gérer elle même** (grâce à ses méthodes internes).

- D'un point de vue conceptuel, appeler une méthode **getXxx**() sur un objet revient à lui envoyer un **message** (ou ***requête***) du genre "***retourne moi la valeur courante de ta propriété Xxx***" .

Exemple (2 versions d'une classe Rectange):

```
public class Rectangle {
        private int x1;
        private int y1;

        /* private int largeur; // v1 */
        private int x2; // v2
        /* private int hauteur; // v1 */
        private int y2; // V2

        public int getX1() {return x1;}
        public void setX1(int x1) {  this.x1 = x1;}

        public int getY1() {return y1; }
        public void setY1(int y1) { this.y1 = y1;}

        public int getLargeur() { /* return largeur; // V1 */      return (x2-x1); // V2  }

        public void setLargeur(int largeur) { /* this.largeur = largeur; // V1 */
                this.x2 = this.x1 + largeur; // V2  }

        public int getHauteur() { /* return hauteur; //v1 */     return (y2-y1); // V2 }

        public void setHauteur(int hauteur) { /* this.hauteur = hauteur; // V1 */
                this.y2 = this.y1 + hauteur;  // V2}
}
```

==> Ces 2 versions sont vues de la même façon de l'extérieur. La différence n'est qu'interne.

4. Ramasse miettes (G.C.)

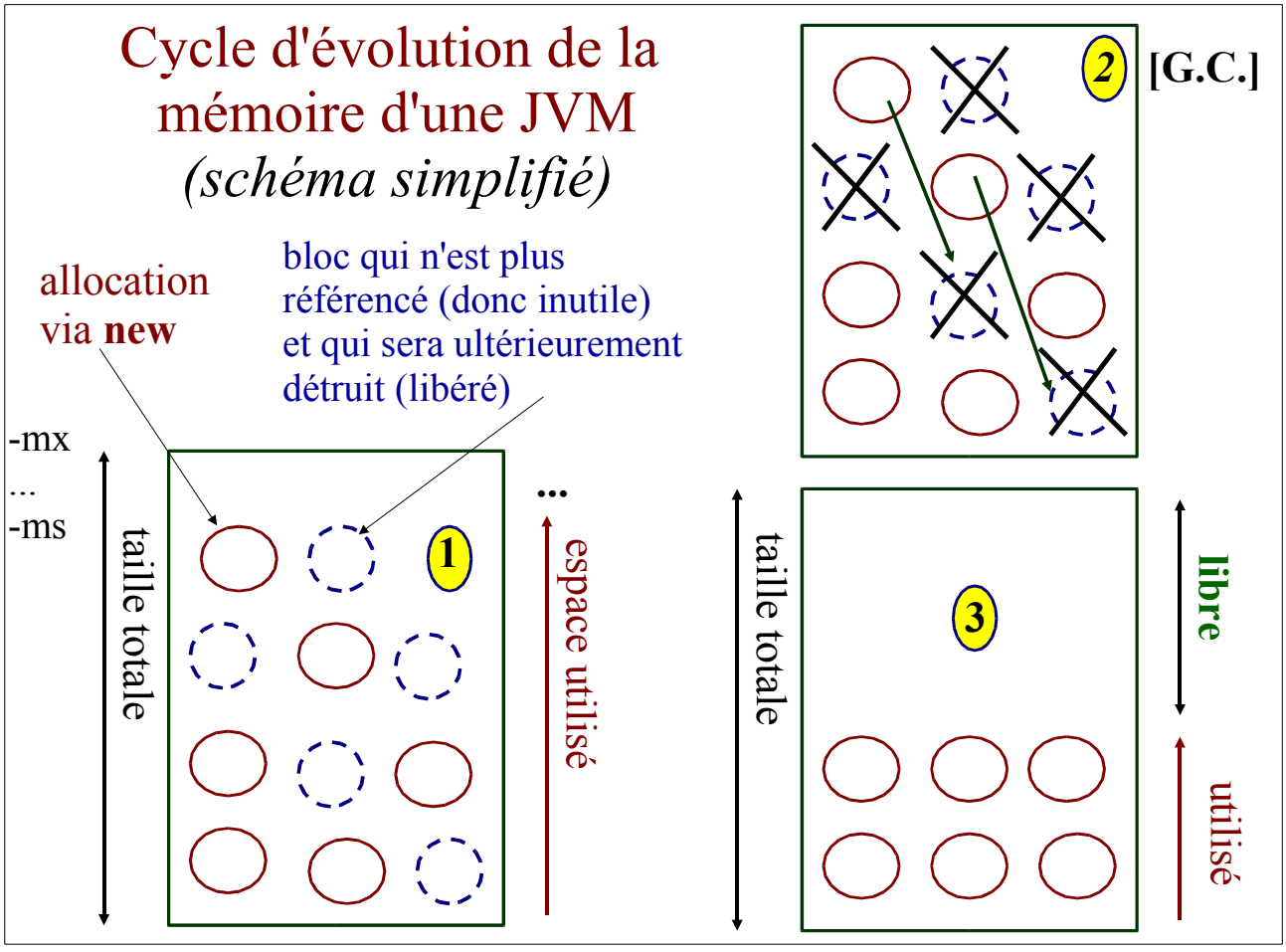
4.1. Destruction automatique des objets

Contrairement aux langages C/C++, le langage JAVA ne comporte ***pas de mot clef*** *delete* (ni *free*) pour libérer la mémoire dynamiquement allouée par les précédents appels à ***new***.

Les objets JAVA sont automatiquement détruits par un mécanisme interne de la machine virtuelle dès qu'ils ne sont plus référencés. Le programmeur n'a donc plus à gérer explicitement la désallocation de la mémoire: il n'a plus qu'à oublier les objets dont il n'a plus besoin.

Le **ramasse-miettes JAVA** *(appelé **Garbage Collector** en anglais)* est exécuté par un thread de basse priorité tournant en tâche de fond.

Règle importante: **un objet JAVA sera** habituellement détruit **lorsque toutes les références sur celui-ci** (variables locales, membres d'autres objets, …) **auront disparues**.



=> **Une défragmentation de la mémoire est ainsi régulièrement effectuée**.

NB1: pour optimiser la gestion de la mémoire, on peut explicitement indiquer à JAVA que l'on a plus besoin d'un objet en donnant la valeur ***null*** à une référence sur celui-ci:

```
Compte  c= new Compte();
c.solde = 120.5;
...
c = null; // ce petit coup de main donné au ramasse miettes n'a d'intérêt que si c peut exister longtemps
          // si la référence c est une variable locale, elle sera rapidement détruite en fin de fonction.
...
```

NB2: **System.gc()** permet de demander au ramasse miettes de s'exécuter tout de suite.

Cependant, ce ramasse-miettes se met à travailler que s'il y a au moins n octets à libérer en mémoire. On est donc jamais complètement sûr du moment où les objets sont détruits.

4.2. Finalisation d'objets

Symétriquement aux constructeurs, les ***finaliseurs*** sont des ***méthodes qui sont automatiquement appelées juste avant qu'un objet d'une certaine classe soit détruit***.

De la même façon que les destructeurs du langage C++, **les finaliseurs du langage JAVA ne servent pas à détruire l'objet lui même mais servent à déclencher une cascade d'autres actions à répercuter** *(fichiers à fermer, déconnexions ...)* **juste avant que celui-ci ne soit détruit**.

Le finaliseur d'une classe JAVA est une méthode qui porte obligatoirement le nom "***finalize***".
Exemple:

```
…
protected void finalize() throws IOException
    { if(...)
        fic.close();
    }
…
```

Remarque très importante:

- Etant donné qu'on ne contrôle pas du tout le moment exact où un objet java sera détruit, **il faut absolument appeler nous même** *(dans le cadre d'une bonne programmation)* **les instructions de fermeture** *.close()* **dans le bon ordre :** *ordre inverse des ouvertures / connexions* .

- Une méthode finalize() appelant une méthode *close()* ne doit être considérée que comme un mécanisme permettant de remédier à des oublis .

5. Opérateurs du langage Java

5.1. Liste des opérateurs du langage JAVA:

opérateurs	*Utilisations*	*Sémantiques*
+	expr + expr	Addition
-	expr - expr/ -expr	Soustraction / Opposé
*	expr * expr	Multiplication
/	expr / expr	Division
%	expr % expr	Modulo (reste de la division entière)
^	expr ^ expr	Ou exclusif *(1 ^ 1 ==> 0)*
&	expr & expr	Et bit à bit
|	expr | expr	Ou inclusif bit à bit
!	! expr	négation logique
=	lvalue = expr	Affectation
<	expr < expr	inférieur
>	expr > expr	supérieur
<=	expr <= expr	inférieur ou égal
>=	expr >= expr	supérieur ou égal
++	++expr / expr++	incrémentation (pré/post)
--	--expr / expr--	décrémentation (pré/post)
<<	expr << expr	décalage à gauche
>>	expr >> expr	décalage à droite
>>>	expr >>> expr	décalage à droite *(avec introduction de **0**)*
instanceof	if(obj instanceof classeXXX) …	test appartenance à une classe
==	expr == expr	(test) égalité (res. booléen)
!=	expr != expr	(test) différence
&&	expr && expr	Et logique
||	expr || expr	Ou logique
+=	lvalue += expr	équivalent à lvalue = lvalue + expr
-=	lvalue -= expr	équivalent à lvalue = lvalue - expr
*=	lvalue *= expr	équivalent à lvalue = lvalue * expr
/=	lvalue /= expr	équivalent à lvalue = lvalue / expr
%=	lvalue %= expr	équivalent à lvalue = lvalue %expr
^=	lvalue ^= expr	équivalent à lvalue = lvalue ^ expr
&=	lvalue &= expr	équivalent à lvalue =lvalue & expr
|=	lvalue |= expr	équivalent à lvalue = lvalue | expr
<<=	lvalue <<= expr	équivalent à lvalue = lvalue<< expr
>>=	lvalue >>= expr	équivalent à lvalue = lvalue>> expr
()	(type) expr	conversion de type / casting

NB: y=Math.abs(**x++**) s'exécute avec la valeur courante de x (avant incrémentation).
Si x valait initialement 5 ===> y = 5 et x=6
y=Math.abs(**++x**) s'exécute avec la valeur préalablement incrémentée de x
Si x valait initialement 5 ===> y = 6 et x=6

Les opérateurs "bit à bit" opèrent sur les représentations binaires des nombres entiers:

a = 5 (= $1*2^2 + 0*2^1 + 1*2^0$) se code en binaire comme **0...0000101**
b = 3 (= $1*2^1 + 1*2^0$) se code en binaire comme **0...0000011**

c = a & b donne donc **0...0000011** soit **c=1**
c = a | b donne donc **0...0000111** soit **c=7**
c = a ^ b donne donc **0...0000110** soit **c=4**
c = a << 3 donne donc **0...0101000** soit **c=5 * 2^3 = 40**

Le résultat d'un ET logique est globalement faux si la première expression est fausse ==> la seconde expression ne sera donc évaluée que si la première est vraie :
if(ch !=null *&&* ch.equals("ok")) ...

Attention à ne pas confondre l'affectation (=) avec un test d'égalité (==)

5.2. Priorités des opérateurs

Niveau de priorité	*opérateurs*	*associativité*
1	! - *unaire* ++ -- (*type*)	droite-->gauche
2	* / %	gauche-->droite
3	+ -	gauche-->droite
4	<< >> >>>	gauche-->droite
5	< > <= >= instanceof	gauche-->droite
6	== !=	gauche-->droite
7	& *bit à bit ou booléen*	gauche-->droite
8	^ *bit à bit ou booléen*	gauche-->droite
9	| *bit à bit ou booléen*	gauche-->droite
10	&& *booléen*	gauche-->droite
11	|| *booléen*	gauche-->droite
12	?: *(Expression conditionnelle si?alors:sinon)*	droite-->gauche
13	= *= /= += -= %= <<= >>= &= ^= >>>=	droite-->gauche

NB: l'**associativité** désigne l'ordre d'enchaînement lorsqu'il n'y pas de parenthèse:

32 / 2 / 4 s'exécute comme ***(32/2) / 4*** soit de gauche à droite

a = b =c s'exécute comme ***(a= (b=c))*** soit de droite à gauche

5.3. Opérateur "instanceof"

```
java.util.ArrayList liste = new java.util.ArrayList();
liste.add(new String("abc"));
liste.add(new Integer(5));
```

...

```
java.util.Iterator it = liste.iterator();
Object element = it.next();   // it.next(); retourne une chose vague de type Object
...
if( element instanceof String) // si l'élément est de type String (?)
 {
// Demander l'environnement java de considérer l'élément
// comme étant de type précis "String"  (conversion - casting) :
String chElement = (String) element;

// Appeler une méthode (fonction interne) spécifique à la classe "String"
// (n'existant pas au niveau de la classe générique "Object"):
System.out.println("Le premier caractère est " + chElement.charAt(0) );
 }
```

6. Boucles & instructions de Java

6.1. tests conditionnels (if)

```
if ( x > 0)
      y=x; /* une seule instruction qui n'a pas besoin d'être englobée par des {} */
  else   y=-x;
```

```
if( a == b) /* jamais de then , ce mot clé n'existe pas , la sémantique est implicite */
      {  s = a*a ; p = 4*a ;  }
  else
      {  s = a*b ; p = 2*(a+b) ; }
```

6.2. branchements à choix multiples (switch/case)

```
switch(n)   /* n doit être d'un type primitif exact (ex: int, char mais pas double ) */
            /* depuis le jdk 1.7 , le switch/case fonctionne également avec des "String" */
{
case 0:
    System.out.println("Choix 0"); break;
case 5:
case 10:
    System.out.println("Choix 5 ou 10"); break;
  …
default: System.out.println("Choix par défaut");
}
```

6.3. Boucle while (tant que ...)

```
int i = 100;
while (i > 0) /* tant que */
 {
 … ; i-- ;
 }
```

```
do /* faire au moins une fois , …*/
{
 … } while (i != 0); /* répéter tant que */
```

6.4. Boucle for (*pour i allant de ... à ...par pas de ...*)

```
for(int i=0 /* valeur initiale du compteur */ ; i<n /* teste d'arrêt */ ; i++ /* ou i = i+ 1*/ )
  {   System.out.println(i);
      /* instruction(s) exécutées avant l'incrémentation (i++) */     }
```

L'instruction **break** sert à déclencher une **sortie anticipée de boucle** :

```
for(i=0;i<n;i++)   {   if(tab[i] == valRecherchee)
                                  { indice=i; break; }
                   }
```

NB: l'instruction **continue** permet **passer directement à l'itération suivante** en sautant toutes les instructions situées entre le mot clef continue et l'accolade fermante de la boucle.

7. Chaînes de caractères

7.1. char, String , StringBuffer

Le langage Java comporte 3 types de données permettant de gérer les caractères et les chaînes de caractères:

Types	*caractéristiques*	*valeurs littérales*
char	**caractère UNICODE** (codé sur 2 octets)	**'A'** , **'\u**0041'
String	**Chaîne de caractères** (instance représentant la valeur globale de toute la chaîne).	**"abc"**
StringBuffer	***Buffer de caractères*** servant à construire efficacement une chaîne via de multiples **concaténations**.	*à convertir en String*

L'opérateur + permet d'effectuer des **concaténations** entre 2 objets de type **String**.

```
String ch1="ile" , ch2=" de ", ch3="Ré", ch4=null;
ch4 = ch1 + ch2 + ch3; // concaténation ==> "ile de Ré"
```

Principales méthodes de la classe String :

.**charAt**(i) *avec i entre **0** et **n-1***	retourne le i^{éme} caractère de la chaîne
.**length**()	retourne la longueur de la chaîne
.**equals**(autreChaine)	test si la chaîne courante à la même valeur qu'une autre
.**substring**(firstPos,lastPos+1);	retourne une nouvelle instance = sous partie de la chaîne.

Exemple:

```
String chFileName = "ficA.txt";
int n = chFileName.length(); // 8 caractères
String chExt = chFileName.substring(n-3,n);
```

```
String ch="";
for(int i=0;i<64;i++)
        ch=ch + "*";
//Cette version n'est pas bien (performante) car chaque itération de la boucle for
//implique la création d'un nouvel objet de type String (résultat de la concaténation)
//et la destruction (différée) de l'ancienne chaîne référencée par ch .
```

```
StringBuffer buffer = new StringBuffer(64); // Bien !!!
for(int i=0;i<64;i++)
        buffer.append("*");
String ch = buffer.toString();
```

Quelques autres méthodes disponibles sur la classe String:

indexOf(String chMotif ...)	retourne la première position où la sous-chaîne est trouvée
toLowerCase()	retourne une nouvelle instance en *minuscules*
toUpperCase()	retourne une nouvelle instance en *MAJUSCULES*
String.valueOf(...)	convertit une valeur de type *int , double , ...* en *String*

7.2. Expressions régulières

Depuis la version 1.4 du JDK, la classe **String** comporte certaines méthodes permettant de gérer efficacement les **expressions régulières**.

Rappels sur les expressions régulières:

.	caractère quelconque
.* , c*	* signifie 0 ou n fois le caractère précédent (.* ==> chaîne quelconque)
[0-9] , [a-z]	un caractère au sein d'une plage possible

Gestion des expressions régulières depuis la classe String:

```
String ligne="1969";
String grep_regexp="[0-9]*";
if(ligne.matches(grep_regexp))
        ....
```

String newLigne = ligne.**replaceAll**(replace_regexp,replace_text);

8. Tableaux

8.1. Syntaxe liée aux tableaux

Au sein du langage Java, Les **tableaux** sont manipulés comme des **objets assez particuliers:**

- En tant qu'objets, ils sont créés dynamiquement (via new) puis manipulés au moyen de références.
- En tant que tableaux , on peut utiliser l'**opérateur []** pour directement accéder au **i^{ème} élément**.

***Syntaxe**:*

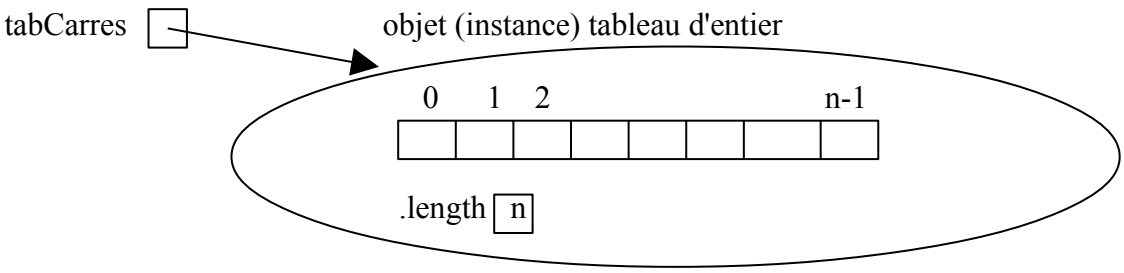
```
TypeElt [ ] refSurTableau=null;  // ou bien  TypeElt refSurTableau[]=null;
...
refSurTableau = new TypeElt[tailleDuTableau];
...
refSurTableau[positionElt] = valeur;
```

***Exemple**:*

```
byte[] buffer;  // ou bien byte buffer[];   ---  déclaration d'une référence sur un futur tableau

buffer =new byte[1024]; // construction d'un tableau de 1024 éléments (taille fixe)

buffer[7]=0; // mettre la valeur 0 dans la (7+1) ème case du tableau.
```

// NB: tout tableau a un champ **length** correspondant à sa taille:

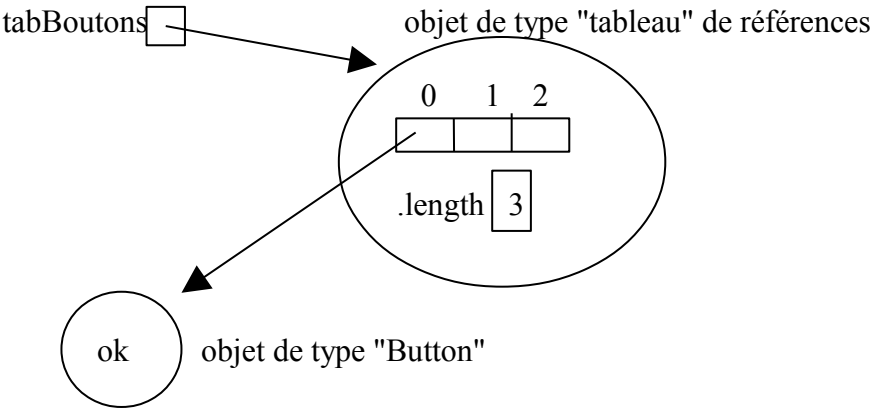
```
int[] tabCarres = new int[20]; // déclaration et construction possible sur la même ligne

for(int i= 0; i<tabCarres.length; i++)
        tabCarres[i] = i * i;
```




```
int tableau_en_dur[] = { 1 , 2 , 4 , 8, 16 };  // initialiseur statique
```

tableau de références sur des objets:

Button[] tabBoutons = new Button[3]; *// tableaux de 3 références non initialisées.*
tabBoutons[0] = new Button("ok"); *// la première case du tableau référence un bouton*
…



Nb:

Toute tentative d'accès à un élément dont l'index est hors de l'intervalle ***[0,tableau.length-1 [*** provoquera la levée d'une exception de type ***ArrayIndexOutOfBoundsException***.

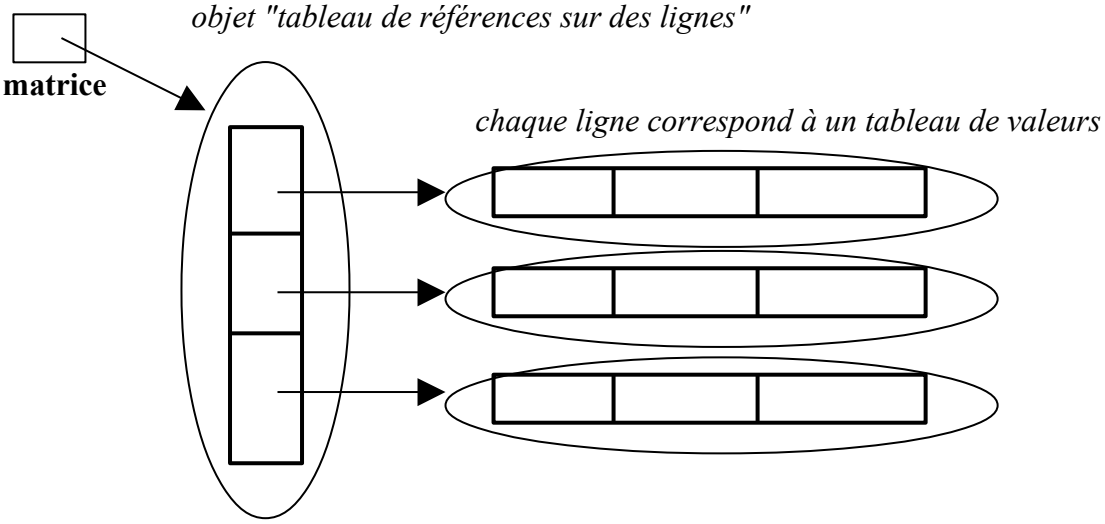
Remarques importantes:

- Un **tableau ordinaire** a une **taille** qui est **fixée de façon rigide** au moment de sa création en mémoire (lors du ***new***). A l'inverse une collection de type **Vector** ou **ArrayList** (du package *java.util*) a le mérite d'être **redimensionnable.**

- Un **tableau ordinaire peut comporter des éléments de types élémentaires** *[non orientés objet]* (int , double , boolean, ...) et **tous les éléments d'un tableau sont issus d'un même et unique type**.
  Les classes **Vector** et **ArrayList** (du package java.util) et toutes les autres collections **ne peuvent comporter que des références sur des objets** (String , Integer mais pas int).

8.2. Tableaux multi-dimensionnels

Java implémente , les tableaux à plusieurs dimensions sous forme de tableau de références sur des sous tableaux.



On peut alors soit:

- allouer toutes les dimensions au départ (***double[][][] matrice = new double[3][3];***)
- allouer que les premières dimensions (***double[][][] triangle = new double[12][];***)
  pour allouer ultérieurement les autres dimensions:
  for(i=0;i<12;i++) triangle[i] = new double[i+1];

On peut également initialiser en dur des tableaux multi dimensionnels:

```
int[][] triangle2 = { {1} , { 1, 2 } , { 1 , 2 , 4 } , { 1, 2,4,8} };

String[][] param_info = { { "param1", "type1" , "libelle1" },
                          { "param2", "type2" , "libelle2" }   }
```

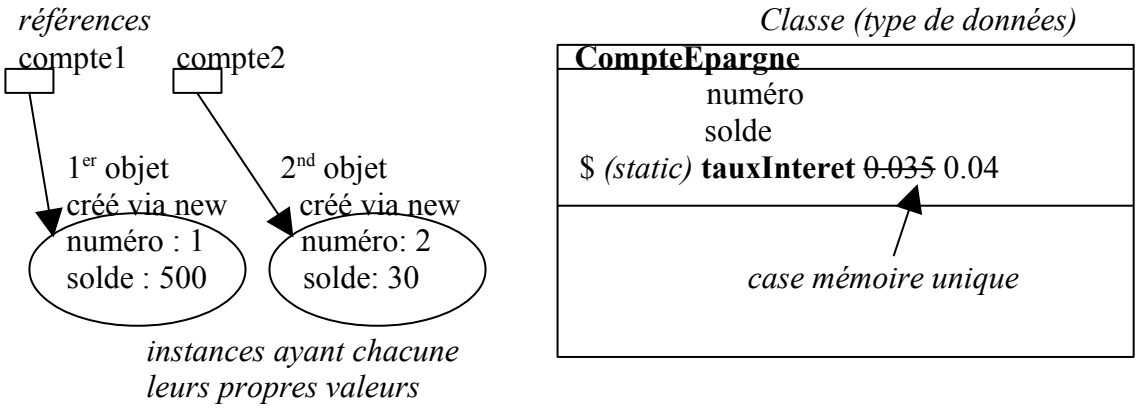
9. Méthode et variables de classes

9.1. Variables de classe

Le mot clé **static** permet de définir des **attributs particuliers**, appelés **"variable de classe"**, qui seront **partagés par toutes les instances d'une même classe**. On aura alors affaire à une **case mémoire unique qui sera liée à la classe plutôt qu'à chacune des différentes instances**.

exemple:

```
public class CompteEpargne {
private String  numero;
private double solde;
private static double tauxInteret = 0.035;
...
public CompteEpargne(String num,double soldeInitial)
   { numero = num; solde= soldeInitial; }
...
public static double getTauxInteret() { return tauxInteret; }
public static void setTauxInteret(double taux) { tauxInteret = taux; }
}
```




```
compte1.setNuméro(1);

compte1.tauxInteret = 0.04; // possible (depuis même package) mais très déconseillé
compte1.setTauxInteret(0.04); // possible mais très déconseillé(ambigu).
CompteEpargne.tauxInteret = 0.04; // c'est un peu mieux(possible depuis même package)
CompteEpargne.setTauInteret(0.04); // c'est encore mieux
```

Autre exemple:

```
public class CXxx {
/* public */   static int nb_instances = 0;  // compteur d'instances
…
public CXxx () { nb_instances++;    ... } // constructeur(s)
protected void finalize() { nb_instances--;    ...}
}
```

Une **variable de classe** *(statique)* peut naturellement être **préfixée par un nom de classe**:

System.out.println("Nombre d'objets créés: " + **CXxx.nb_instances**);

Les **variables de classes publiques** forment **le plus proche équivalent JAVA des variables globales du langage C/C++** .

9.2. Constantes

Au sein du langage JAVA, une **constante** ne peut se définir que sous la forme d'une **variable de classe déclarée finale** *(ne pouvant plus changer de valeur):*

```
public class Cercle
{
public static final double PI = 3.141592653589… ;
… }
```

On pourra ainsi écrire ***2 * Cercle.PI * r*** .

NB: La classe **Math** contient déjà une constante dénommée **PI**.
Par convention, toutes les constantes sont déclarées en MAJUSCULES.

9.3. Méthodes de classe

Le mot clé **static** permet également de définir une **méthode de classe** .
Une telle **méthode statique** a la particularité de **pouvoir être invoquée depuis un nom de classe** et non pas seulement depuis une instance particulière de la classe.

Exemples:

```
public class CompteEpargne {
...
private static double tauxInteret = 0.035;
...
public static double getTauxInteret()  { return tauxInteret; }
public static void setTauxInteret(double taux)  { tauxInteret = taux; }
}
```

```
CompteEpargne c1,c2;
c1=new CompteEpargne("101",100);
c2=new CompteEpargne ("102",200);
double tauxInteretCourant = CompteEpargne.getTauxInteret();
```

```
// Appel de la méthode de classe "Math.sqrt"
double distance = Math.sqrt(x*x+y*y);
```

NB:

- Les méthodes statiques sont en JAVA ce qui se rapproche le plus des fonctions globales du langage C.
- Etant donné qu'une méthode statique est généralement appelée depuis une classe et non pas à partir d'une instance particulière, le mot clé **this** ne peut pas être utilisé au sein du code interne d'une méthode statique.
- La plus classique des méthodes statiques est la fameuse méthode principale **main()**.

Exemple:

```
public class MyApp {

public static void main(String[] args)
        {
         System.out.println("Démarrage du programme \n");
         }
}
```

9.4. Initialiseurs statiques

Un constructeur ordinaire est appelé au moment où un nouvel objet est construit et ne peut initialiser que des variables d'instances (attributs qui ne sont pas "*static*").

Les variables de classes sont quant à elles construites lors du chargement de la classe en mémoire. Elles peuvent être initialisées en utilisant ce qu'on appelle un "***initialiseur statique***".
Il s'agit d'un ***bloc de code (entre { }) directement préfixé par le mot clé static***.

Exemple:

```
public class CXxx {
        …
         private static int  tabCoeff[] = new int[4];

        …
        static {
                tabCoeff[0]=2;  tabCoeff[1]=4;  tabCoeff[2]=1;  tabCoeff[3]=2;
        }
…
}
```

III - Héritage , polymorphisme , interface

1. Généralisation / Héritage

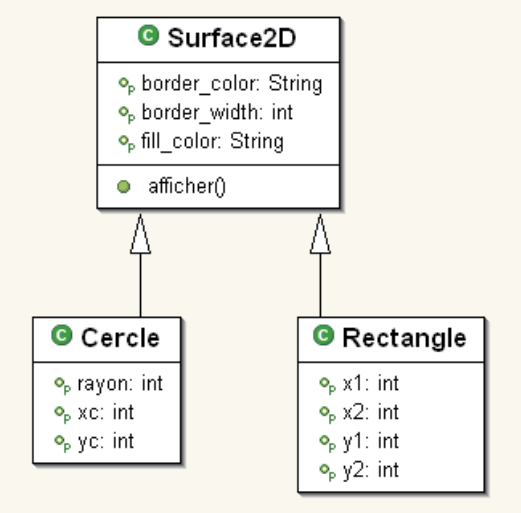
Lorsque l'on souhaite définir une **nouvelle classe dont une partie est déjà implémentée par une classe de base existante** et qui d'autre part doit comporter de **nouvelles spécificités** , il suffit alors de la faire **hériter** de la classe de base.

Dans le schéma conceptuel objet , l'**héritage correspond à un ajout de spécifications**. On parle de **spécialisation** (inversement vu comme une **généralisation**) .Pour que l'héritage ait une signification correcte, on doit absolument pouvoir mentionner :

classe dérivée (sous classe)	**est une sorte de**	***classe de base (super classe)***

Remarque:

- Les relations d'héritage sont souvent représentées sous la forme d'un arbre.
- En UML, la relation **est une sorte de** est représentée par une **flèche** allant de la ***classe dérivée (sous classe)*** vers la ***classe de base (super classe)***.



1.1. Syntaxe de l'héritage en JAVA:

```
public class NomClasseDérivée   extends   NomSurClasse
{
... // Attributs et méthodes supplémentaires
...// Opérations (Méthodes) redéfinies (même signature, nouveau code)
}
```

NB:

- Quand une classe entière est déclarée avec le mot clé ***final***, elle ne peut plus être étendue (on ne peut plus en hériter). Ceci permet au compilateur d'effectuer quelques optimisations. *java.lang.System* constitue un exemple de classe finale.

- Si on ne définit pas de super-classe au moyen de la clause **extends**, le langage JAVA fournit une ***sur-classe par défaut*** appelé **Object**.

- La classe **Object** est ainsi une classe très spéciale dont toutes les autres classes JAVA héritent directement ou indirectement. La classe **Object** constitue ***le sommet de l'arbre d'héritage*** en JAVA.

Nouveau mot clef partiellement lié à la notion d'héritage: **protected** (protégé)

Tout comme private et public , le qualificatif **protected** permet de définir la **visibilité des membres d'une classe. Les membres protégés seront accessibles** depuis les méthodes de la classe courante ainsi que **depuis toutes les méthodes des classes dérivées appartenant éventuellement à d'autres package** (sous (sous) classes) **mais resteront inaccessibles depuis l'extérieur** (autres classes et autre package).

Exemple:

```
public class Surface2D   /* classe de base */
{
        protected String fill_color;
        protected String border_color;
        protected int border_width;

        public Surface2D() { border_width=1; border_color="black"; fill_color="white"; }

        public String getFill_color() { return fill_color; }
        public void setFill_color(String fill_color) { this.fill_color = fill_color; }
        public String getBorder_color() { return border_color; }
        public void setBorder_color(String border_color) { this.border_color = border_color; }
        public int getBorder_width() { return border_width;}
        public void setBorder_width(int border_width) { this.border_width = border_width; }

        public void afficher() {
                System.out.println("bordure de couleur " + border_color +
                                   "et d'épaisseur " + border_width );
                System.out.println("couleur de remplissage = " + fill_color );
        }
}
```

```
public class Cercle extends Surface2D    /* Cercle hérite de Surface2D */
{
        private int xc;
        private int yc;
        private int rayon;
```

```
        public int getXc() { return xc; }
        public void setXc(int xc) { this.xc = xc; }

        public int getYc() { return yc; }
        public void setYc(int yc) { this.yc = yc; }

        public int getRayon() { return rayon; }
        public void setRayon(int rayon) { this.rayon = rayon; }

        public void afficher() {
                System.out.println("Cercle de centre (" + xc + "," + yc +
                                        ") et de rayon " + rayon );
                super.afficher();
        }
        ...
}
```

1.2. Utilisation d'une sous classe

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

        Surface2D s = null;  //référence sur surface quelconque
        Cercle c =null; // référence sur cercle quelconque

        c=new Cercle(); // instanciation de la classe dérivée

        int ep = c.getBorder_width(); // appel direct d'une méthode héritée

        c.setXc(23); c.setYc(67);
        c.setRayon(234); // appel de l'une des méthodes supplémentaires

        c.afficher(); // appel de la nouvelle version de afficher (spécifique aux Cercles)

        s= c; // s référence un cercle  qui est un surface particulière.
        String couleur_remplissage = s.getFill_color();

        /* s.setRayon(15); */     // fonction uniquement valable
        ((Cercle) s).setRayon(15);  //sur le type de donnée Cercle

        s.afficher(); // tient compte de la nature exacte de l'objet référencé (polymorphisme).
        }
}
```

1.3. Liaison entre une méthode redéfinie et la version héritée

Le mot clé ***super*** peut servir à appeler, au sein d'une méthode que l'on redéfinit, le code de la version liée à une super-classe.

Exemple:

```
public class Cercle extends Surface2D    /* Cercle hérite de Surface2D */
{                   ...
        public void afficher() {
                System.out.println("Cercle de centre (" + xc + "," + yc +
                                        ") et de rayon " + rayon );
                super.afficher();
        }
}
```

NB:

- Au sein d'une méthode redéfinie qui n'est pas un constructeur, on peut utiliser le mot clé super sur une ligne quelconque (pas obligatoirement la première)
- La construction suivante est illégale : ~~super.super.m1();~~
- Etant donné que les finaliseurs ne sont pas automatiquement enchaînés, on peut explicitement préciser l'enchaînement en incorporant une dernière ligne du type ***super.finalize();***

1.4. Constructeur d'une sous classe

Le constructeur d'une sous-classe peut appeler celui de sa super-classe en utilisant le mot clef **super**.
Exemple:

```
public  class  Surface2D
{ ...
public Surface2D(String couleur_bordure, int epaisseur) //Constructeur
        {
         border_color=couleur_bordure;   border_width = epaisseur;
         }
}
```

```
public class Cercle extends Surface2D
{...
public Cercle(double x, double y, double r,String couleur_bordure, int epaisseur) //Constructeur
        {
        super(couleur_bordure, epaisseur);  // appel au constructeur de la super-classe Surface2D
        this.xc = x; this.yc=y; this.rayon=r;   }
}
```

NB:

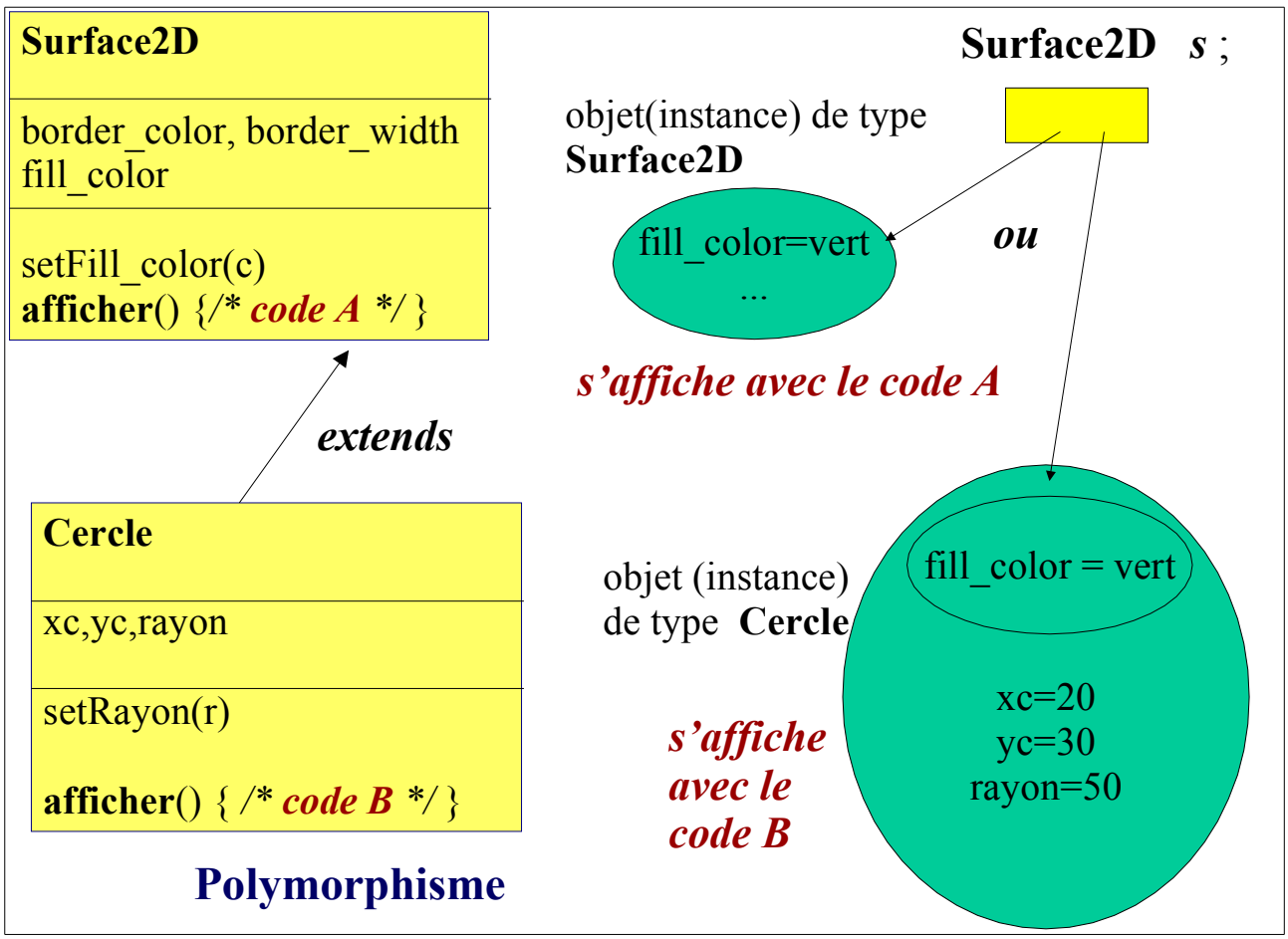
- Le mot clef ***super*** ne peut être utilisé qu'en tant que ***première ligne de code d'un constructeur*** d'une sous-classe.
- Si l'appel à ***super*** n'est pas explicitement mentionné, il est alors implicitement introduit sans argument. Ce qui a pour effet d'appeler le constructeur par défaut (sans argument) de la surclasse.Il y a donc un ***enchaînement automatique des constructeurs*** : Toute construction d'un objet d'une classe dérivée implique l'appel (implicite ou explicite) du constructeur de la classe d'au dessus qui appel à son tour le constructeur de sa surclasse, …

1.5. Redéfinition de méthode & opérations abstraites

Précisions sur quelques notions fondamentales:

- **Toute nouvelle classe java correspond à un type de données**.
- **Le type de données associé à une sous classe est compatible (en tant que cas particulier) avec le type de données générique lié à la super classe**.
- Une fonctionnalité que l'on peut attendre d'un type d'objet correspond à une opération. Une opération est toujours mise en correspondance avec un nom de fonction. **Une opération a une signature précise** (type de retour et paramètres d'entrée bien précisés).
- **Une opération peut éventuellement être codée différemment dans diverses (sous) classes**.
- **La version exacte d'une opération au niveau d'une certaine classe est appelée méthode**. Une méthode est donc associée à du code bien précis.

2. Polymorphisme



- Lorsque la référence **s** pointe sur un objet de type ***Surface2D*** (suite à une instruction du

genre ***s = new Surface2D()*** ou indirectement via une copie de référence), un appel à l'opération ***afficher()*** déclenche le ***code A*** définit au niveau de la classe de base ***Surface2D***.

- Lorsque cette même référence s pointe sur un objet de type ***Cercle*** (via ***s = new Cercle()***), un appel à l'opération ***afficher()*** déclenche le ***code B*** définit au niveau de la classe ***Cercle***.
- Ainsi, une même ligne de code ***s.afficher();*** peut déclencher plusieurs traitements légèrement différents suivant la nature exacte de l'objet qu'il y a au bout de la référence à un instant précis.

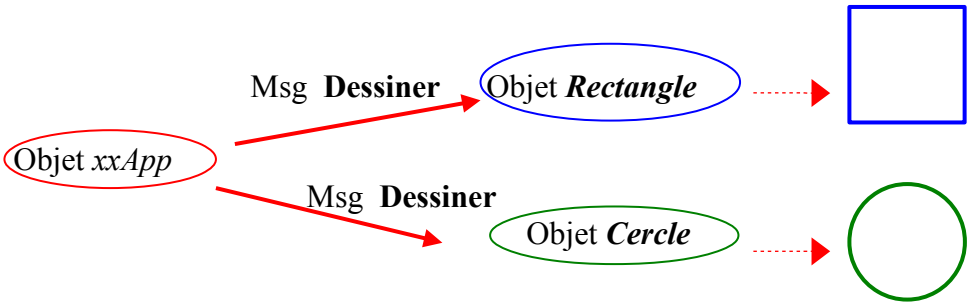
2.1. Liaison dynamique (formalisation du polymorphisme)

Lorsque l'on définit une référence dont le type est une classe de base (générique), cette variable "*référence*" peut très bien référencer une instance d'une sous classe *(s = c de l'exemple ci-dessus)*.

Conceptuellement, Appeler une opération ***afficher*** à partir une référence ***refObj*** (*instruction* ***refObj.afficher();***) consiste à envoyer un message dénommé ***afficher*** *(ou affiche toi)* à l'objet référencé par ***refObj***.

D'après le principe du **polymorphisme** que vérifie implicitement JAVA, on a alors le comportement suivant:

- **Un même message envoyé vers divers objets peut déclencher des actions qui peuvent prendre plusieurs formes différentes**.
- ***L'action (code exécuté) par l'objet (instance) recevant le message dépend de sa classe précise*** (celle qui a été utilisée pour le créer à coup de *new*).



NB: Lorsque le compilateur compile une ligne de code du type ***refObj.afficher();*** il ne connaît pas encore la version exacte de *afficher()* qui sera appelée car ***refObj*** peut référencer des instances issues d'une vaste panoplie de classes. C'est au moment de l'exécution du programme qu'est effectué le choix de la version de la méthode qui sera lancée. On parle de ***liaison dynamique***.

3. Classes abstraites

- Une **classe abstraite** est une **classe intermédiaire dans la partie haute d'un arbre d'héritage** et qui **permet de répertorier un ensemble d'opérations qui seront différemment codées dans les indispensable sous classes concrètes**.

- Une **opération** (méthode) **abstraite** (déclarée avec le mot clé **abstract**) est ***une méthode sans code qui doit absolument être redéfinie dans les sous (sous) classes***.

- ***Une classe contenant au moins une méthode abstraite est elle même abstraite et ne peut pas être directement instanciée*** *(via le mot clef* ***new****)*.

- On peut tout de même déclarer une référence de type abstrait pour ensuite:
  1. référencer des instances issues de sous classes concrètes.
  2. appeler des opérations depuis la référence générique (polymorphisme).

Exemple:

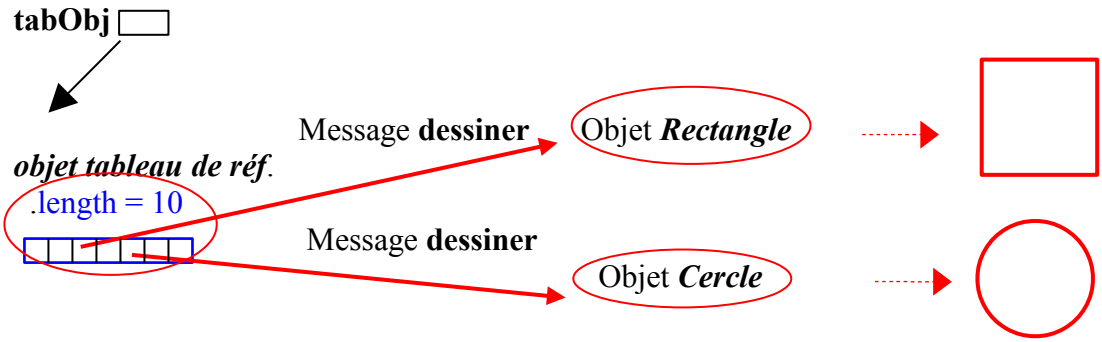
```
public abstract class ObjetGraphique  //classe abstraite
{
private Color couleur; // véritable attribut
public abstract void dessiner(Graphics g);  // opération (méthode) abstraite sans code
public void setCouleur(Color c) { couleur=c; }  // fonction non abstraite (avec code)
public Color getCouleur() { return couleur; }  // fonction non abstraite (avec code)
}
```

```
public class Cercle extends ObjetGraphique  // Cercle = classe concrète
{…
public void dessiner(Graphics g) // méthode redéfinie
                { g.DrawOval(x,y,r,r); }
}
```

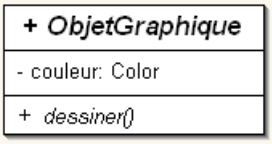
```
public class Rectangle extends ObjetGraphique // Rectangle  = classe concrète
{…
public void dessiner(Graphics g) // méthode redéfinie
                { g.DrawRect(x,y,l,h); }
}
```

```
ObjetGraphique obj; // variable générique pouvant référencer tout type d'objet graphique

obj= new ObjetGraphique(); // new direct impossible car la  classe est abstraite.
```

```
...
Cercle c1 = new Cercle(152.3 /*xc*/ ,15.66 /*yc*/ ,465.2 /*rayon*/);
Rectangle r1 =   new Rectangle( 12.6 /*x1*/, 45.3 /*y1*/, 17.9 /*x2*/, 198.12 /*y2*/);
…
obj = r1;
obj.dessiner(g);  // dessine un rectangle (polymorphisme)
obj = c1;
obj.dessiner(g);  // dessine un cercle (polymorphisme)


ObjetGraphique[] tabObj = new ObjetGraphique[10]; // tableau de 10 références.
tabObj[0] = r1;
tabObj[1] = c1;
…
tabObj[9] = new Cercle(12.5,56.8,45.0);
…
for(int i=0;i<10;i++)
        tabObj[i].dessiner(g); // dessine des rectangles, des cercles , … (polymorphisme)
…
```



- Ce schéma est à reproduire en remplaçant éventuellement le tableau de référence à taille fixe par un vecteur redimensionnable.
- ***Bien que partiellement programmée, une classe abstraite est tout de même vue comme un type de données.***
- Au sein des diagrammes de classes **UML** , les **opérations et classes abstraites** sont représentées en caractères **italiques:**

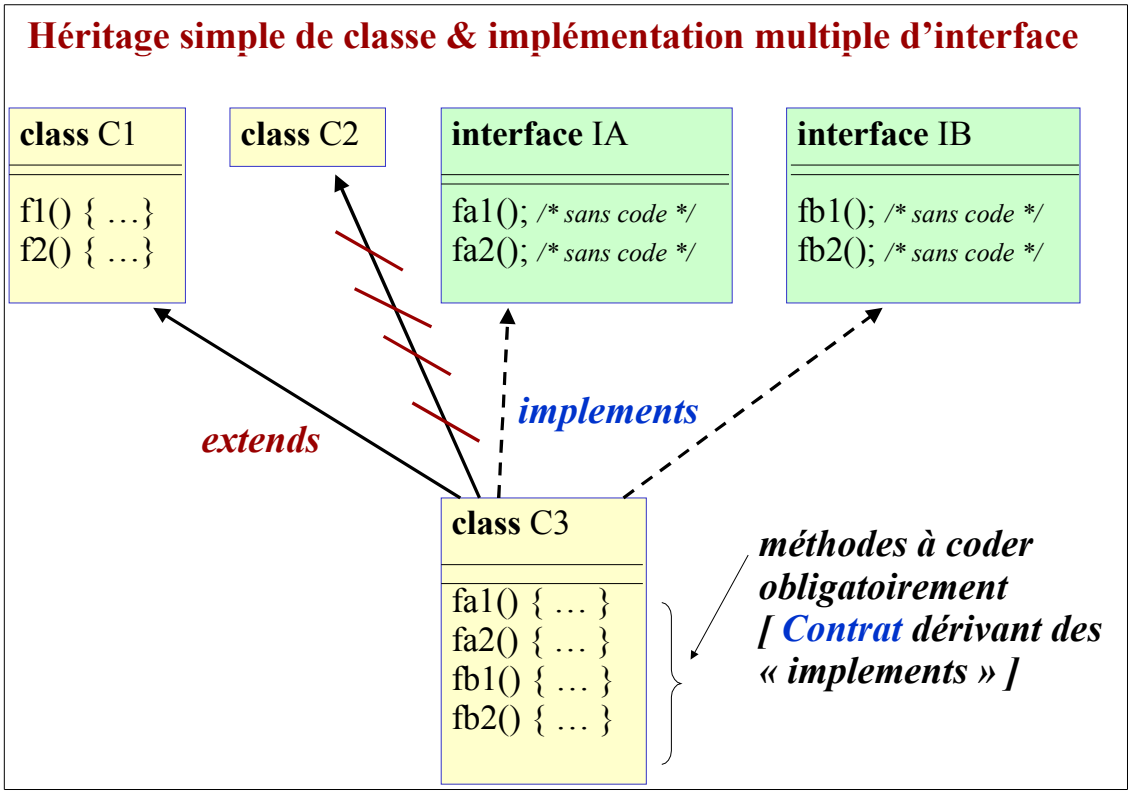


4. Interfaces

4.1. concept d'interface et syntaxe java

Une interface est une pseudo-classe qui ne comporte que des déclarations de méthodes sans code. Une interface ressemble beaucoup à une classe abstraite. ***Toutes les méthodes d'une interface sont implicitement abstraites***.

Une classe JAVA ne peut avoir qu'une seule super-classe. L'héritage multiple d'implémentation n'est pas supporté par JAVA. Le langage ***JAVA supporte par contre un héritage multiple d'interface***. ***Une classe JAVA peut hériter d'une classe concrète et d'un nombre quelconque d'interfaces*** (On dit ***qu'une classe étend une super-classe et implémente certaines interfaces***).



Syntaxe---> public **class** *C3* **extends** *C1* **implements** *IA* **,** *IB*
{ .... }

Exemple:

```
public interface Localisable
{ // méthodes implicitement abstraites:
        public Point  getCenter();
        public int  getWidth();
        public int getHeight();
// constante(s) directement visible(s) (sans préfixe) depuis les classes qui
// implémenteront cette interface :
       public static final  String DEFAULT_UNIT = "mm";
}
```

Nb: une interface publique se programme dans un fichier à part (ayant le même nom que l'interface).

Classe implémentant une interface:

```
public class CercleLocalisable extends Cercle implements Localisable
{ String unite ;
 …
        public CercleLocalisable () { unite = DEFAULT_UNIT; ...}
        public Point  getCenter() { return new Point(xc,yc); }
        public int  getWidth() { return rayon ; }
        public int getHeight() { return rayon ; }
… }
```

NB: **Une classe implémentant une interface est obligée de coder toutes les méthodes abstraites qui en découlent**. Le compilateur Java détecte les oublis et génère des messages d'erreurs .

On peut déclarer une référence générique de type interface XXX, pour ensuite:

- Référencer une instance d'une classe implémentant cette interface.
- Appeler à partir de cette référence, l'une des méthodes redéfinies qui ont été déclarées au niveau de l'interface (*polymorphisme*).

Exemple:

```
Localisable locObj[ ] = new Localisable[10]; // tableau de 10 références
                                             // sur des choses localisables
CercleLocalisable c1 = new CercleLocalisable();
RectangleLocalisable r1 = new RectangleLocalisable();
… locObj[0]=c1;      locObj[1]=r1; …
for(int i= 0;i<10;i++)
     { …  locObj[i].getCenter();  …} // polymorphisme
```

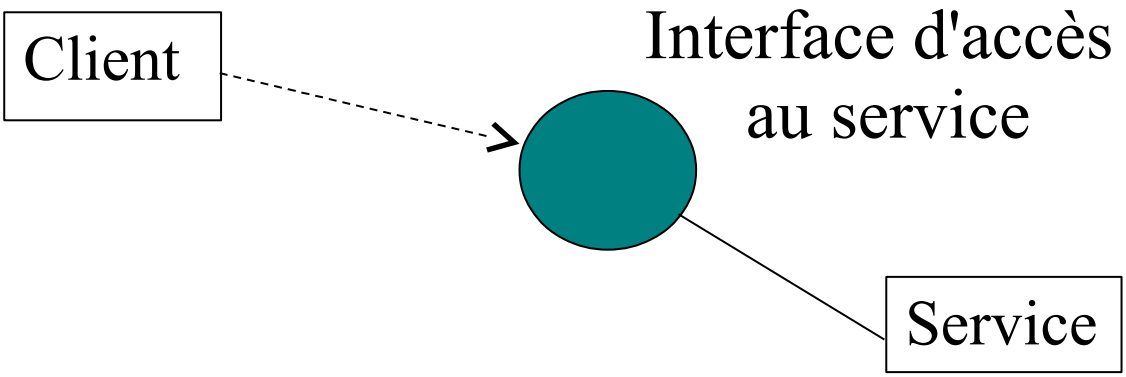
Héritage entre interfaces:

Une interface peut hériter d'une ou plusieurs interfaces existantes.
exemple: public ***interface*** Dynamique ***extends*** Deplacable, Zoomable {}

4.2. Sémantique (signification) des interfaces:

- Une **interface** peut être considérée comme un **contrat** car chaque classe qui choisira d'implémenter (réaliser) l'interface sera obligée de programmer à son niveau toutes les opérations décrites dans l'interface .

- Chacune de ces opérations devra être convenablement codée de façon à rendre le **service effectif** qu'un client est en droit d'attendre lorsqu'il appelle une des méthodes de l'interface.



Grâce à une interface, un morceau de code d'une API prédéfinie pourra activer des traitements au sein de nos propres classes sans préjuger de quoi que ce soit :
Notre classe maison pourra avoir un nom quelconque est pourra hériter de n'importe qu'elle classe. la seule chose imposée est d'implémenter les fonctions de l'interface.

Interface = contrat (liste de méthodes à coder quelque part)

Interface = type de données abstrait

- c'est une des clefs à bien maîtriser pour écrire des programmes modulaires.

==========

La ***logique événementielle*** (qui sera vue dans un chapitre ultérieur) constituera un exemple concret d'utilisation des interfaces .

IV - Eléments structurants de java

1. Packages et archives (.jar)

1.1. correspondance entre nom de package et un chemin relatif

En Java, ***chaque classe compilée est stockée dans un fichier à part***. ***Le nom de ce fichier doit être le même que celui de la classe*** (ex: MaClasse**.class**).

Les classes sont elles mêmes regroupées au sein de "packages".

Le nom d'un package peut être composé et doit être apparenté à une structure arborescente de répertoires et de sous répertoires :
Ainsi le package
com.worldcompany.utils
correspond au répertoire dont le chemin relatif est
com/worldcompany/utils .

Le nom complet d'une classe java est "***nom_composé_du_package.NomClasse***"

Grâce à ceci, une **machine virtuelle JAVA est ainsi capable de situer** (***relativement*** *à partir d'une base de départ*) **le fichier d'une certaine classe d'un certain package**.

1.2. Archive Java (.jar)

Un programme ou une API Java est généralement constitué d'un grand nombre de classes organisées au sein de différents packages.
De façon à faciliter le déploiement vers d'autres machines, toute cette arborescence est en général packagée au sein d'une archive (fichier ".ZIP" ou ".JAR") .

Un fichier **.JAR** (***Java ARchive)*** est un fichier au format **ZIP** (que l'on peut par exemple créer ou consulter avec WinZip ou 7Zip ou bien l'utilitaire bin\jar.exe du jdk).

Construction de **"my_appli.jar"** pour ***démarrer une application via un simple double-click***:

Soit **build.xml** un **script ant** (situé à la racine d'un projet eclipse) et permettant de construire automatiquement une archive "**my_appli.jar**" en fonction de tout le contenu du sous répertoire "***bin***" comportant tout le code compilé (packages + classes java):

build.xml

```
<?xml version="1.0"?>
<project name="my_project" default="default">
    <description>
            script ant
    </description>
```

```
    <!-- =======   target: default ====== -->
    <target name="default" depends="build_jar,generate_javadoc">

    </target>

    <!--    target: build_jar      - - -->
    <target name="build_jar">
      <jar destfile="my_app.jar">
            <manifest>
                  <attribute name="Main-class" value="swing_app.SwingApp"/>
            </manifest>
          <fileset dir="bin">
            </fileset>
      </jar>
    </target>

      <!--  target: generate_javadoc  -->
       <target name="generate_javadoc">
            <javadoc  destdir="docs">
                  <fileset dir="src">
                  </fileset>
            </javadoc>
       </target>

</project>
```

NB: un **script ant** ressemble à un Makefile (il sert à automatiser des compilations, génération d'archives, des déploiements , ...) . Il est encodé en xml et interprété par un programme open source "java"; ce qui lui permet d'être très portable (Windows , Unix , ...).

La cible *build_jar* du **script ant** précédent **(build.xml)** que l'on lance sous **eclipse** via **click_droit / run ant ...** permet de générer une archive "**my_appli.jar**" comportant le fichier ***Manifest*** suivant:

META-INF/Manifest.mf

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.6.2
Created-By: 1.5.0_01-b08 (Sun Microsystems Inc.)
Main-class: swing_app.SwingApp
```

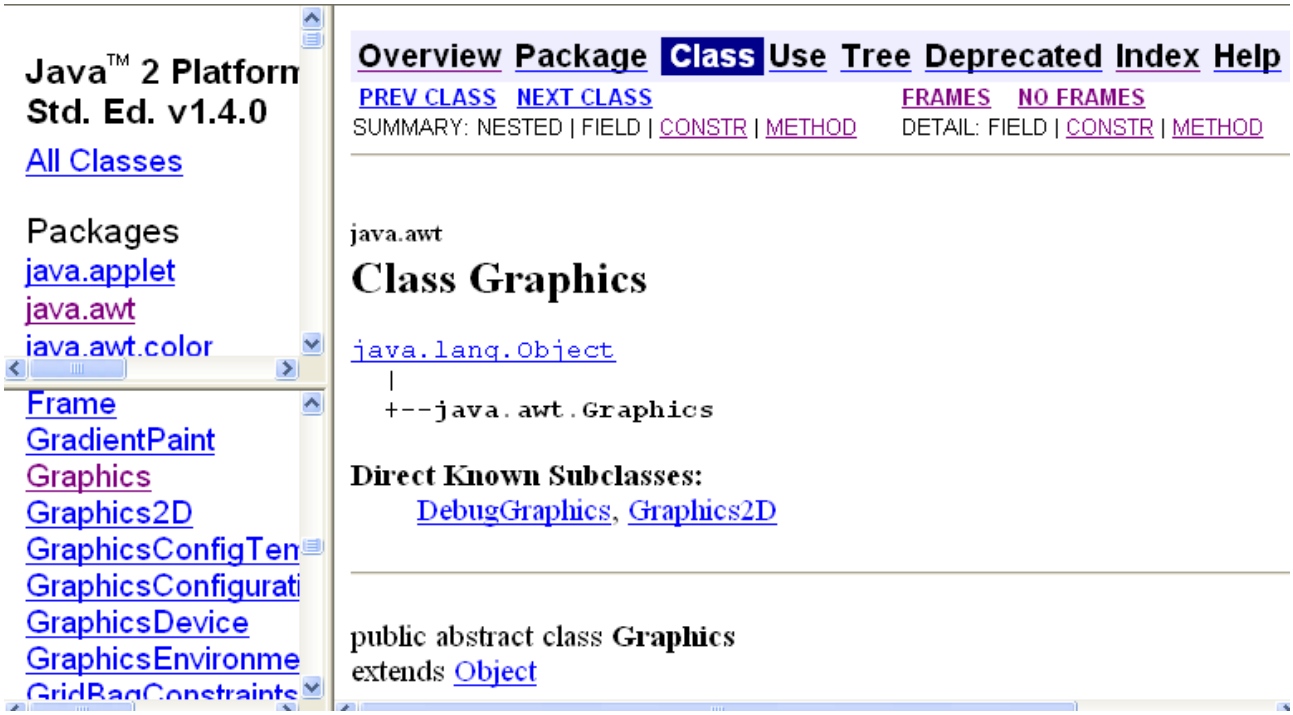
L'indication "*Main-class:* " permet de préciser la classe principale du programme à lancer suite à un double-click sur le "*.jar auto-démarrable*".

La cible *generate_javadoc* du *script ant* précédent permet de **générer automatiquement une documentation HTML** sur les classes du projet .
Le point d'entrée de cette documentation est **docs/index.html** .

NB: Javadoc tient compte des éventuels **commentaires spéciaux** (au format ci-après) pour générer quelques éléments de la documentation:

```
/**
 * @author ddefrance
 *
 * AvionV2 comportant un tableau de références sur des choses descriptibles
 */
....
```

NB: L'aide en ligne sur les éléments prédéfinis du langage java est également générée à partir de javadoc et se consulte de la manière suivante:
docs/index.html (pour les grandes rubriques – présentation générale)
docs/api/index.html (pour les détails des API)



1. Choix d'un package (ex: java.awt)
2. Choix d'une interface ou d'une classe
3. Visualisation des détails (héritages , attributs , méthodes, ...)

1.3. CLASSPATH (liste des endroits où rechercher les classes)

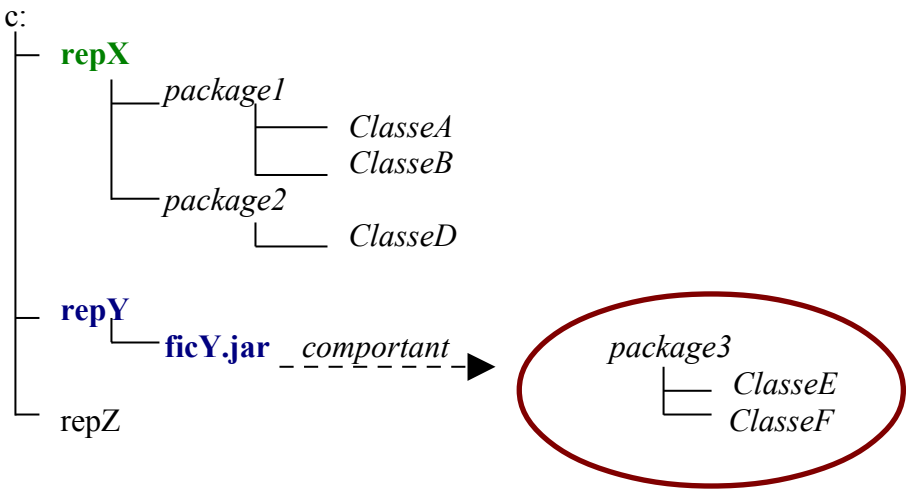
- La variable d'environnement **CLASSPATH** est constituée d'***une liste de chemins menant à des paquets de classes*** que l'on peut avoir besoin de charger en mémoire.

- Chaque **chemin** (séparé par *';'* sous windows ou par *':'* sous unix) est ***soit*** **un répertoire du système de fichier** ***soit*** **une archive** (fichier *'.zip'* ou *'.jar'*) .

- ***Sous chaque partie référencée par le CLASSPATH on doit pouvoir trouver en relatif les sous répertoires associés à une arborescence des packages*** ainsi que les fichiers de classes (.class)

Exemple:

```
set CLASSPATH=%CLASSPATH%;c:\repX;c:\repY\ficY.jar
```



Remarque:
Les classes prédéfinies des API standards de Java sont placées dans l'archive ...\jre\lib**rt.jar** que la machine virtuelle consulte systématiquement.

L'API Java est elle même constituée de plusieurs packages:

Nom du package	***Contenu***
java.applet	Classes pour coder les applets
java.awt	(Abstract Window Toolkit) Gestion des fenêtres, GUI
javax.swing	Classes graphique 100% java (JButton, ...)
java.sql	Accès aux bases de données via JDBC
java.io	Entrées/sorties
java.lang	Base du langage JAVA (types de base, …)
java.net	Communication réseau (Socket, UrlConnection,...)
java.util	Classes utilitaires (Vector, Date, ...)
…	…

Les packages ne servent pas uniquement à trouver les classes à charger.
Les deux grands autres intérêts des packages sont les suivants:

- **Eviter des conflits de noms entre classes développées par différents développeurs**.
  Le nom complet d'une classe est en effet quelquechose du type
  nompackage.NomClasse (ex: ***java.util.Date*** *est différent de* ***java.sql.Date***)

- Délimiter un espace sur lequel certains mots clés du langage (public, private, protected) auront une incidence sur la visibilité des choses.

Les principales règles liées à la visibilité des entités de JAVA sont les suivantes:

- Un package est accessible si les répertoires et fichiers associés le sont.
- ***Toute classe d'un package est accessible depuis toutes les autres classes du même package***.
- ***Seules les classes déclarées publiques*** dans un package ***sont accessibles depuis les autres packages***.
- ***Un membre d'une classe*** (variable ou méthode***) est accessible depuis une autre classe du même package*** si et seulement ***si il n'est pas déclaré privé***. Les champs privés ne sont accessibles qu'à l'intérieur de la classe en question.
- ***Les membres d'une classe sont accessibles depuis un package différent*** à partir du moment où la classe est accessible (public) et **que ses membres sont déclarés publics**, ***ou bien si ses membres sont déclarés protégés(protected) et que l'on y accède depuis une sous classe***.

1.4. Instruction package

L'instruction **package** ne peut apparaître que sur la première ligne (différente d'un commentaire) d'un fichier source.
Cette instruction indique le nom du package auquel appartient le code du fichier source.

```
package com.worldcompany.utils;
```

NB: Si l'instruction package est omise (ce qui n'est pas du tout conseillé), le code du fichier source appartiendra au package par défaut qui n'a pas de nom.

1.5. Instruction import

La directive ***import*** ressemble de loin à l'instruction #include des langages C et C++ .
Vue d'un peu plus près , elle est bien loin d'y ressembler.

La directive **import** indique simplement à JAVA que **l'on pourra utiliser des noms abrégés (sans préfixe correspondant au package) pour nommer certaines classes**.

Exemple :

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

Grâce à l'instruction précédente on pourra écrire :

```
bouton =new Button("Ok");
```

au lieu d'écrire :

```
bouton = new  java.awt.Button("Ok");
```

Remarques importantes:

- L'instruction ***import java.lang.*;*** n'est pas indispensable puisqu'elle est systématiquement et implicitement prise en compte (c'est un cas particulier).

- Le caractère ***** souvent utilisé en fin des directives "**import**" signifie "n'importe quelle classe du package" mais ne signifie pas "n'importe quelle classe et n'importe quel sous package".

```
import  java.awt.*;
import  java.awt.event.*; // les 2 lignes sont nécessaires.
```

- Beaucoup de programmeurs consciencieux préfèrent indiquer **une longue série de directives import** en n'utilisant pas le caractère * mais **en spécifiant à chaque fois le nom exact d'une des classes utilisées**. *[Ceci n'est intéressant qu'au niveau du code source. Le résultat de la compilation est exactement le même]:*

```
package ex;

import java.util.Vector;
import java.util.Iterator;
import java.util.Date;     // On visualise tout de suite toutes les classes utilisées par ce fichier de code

...
```

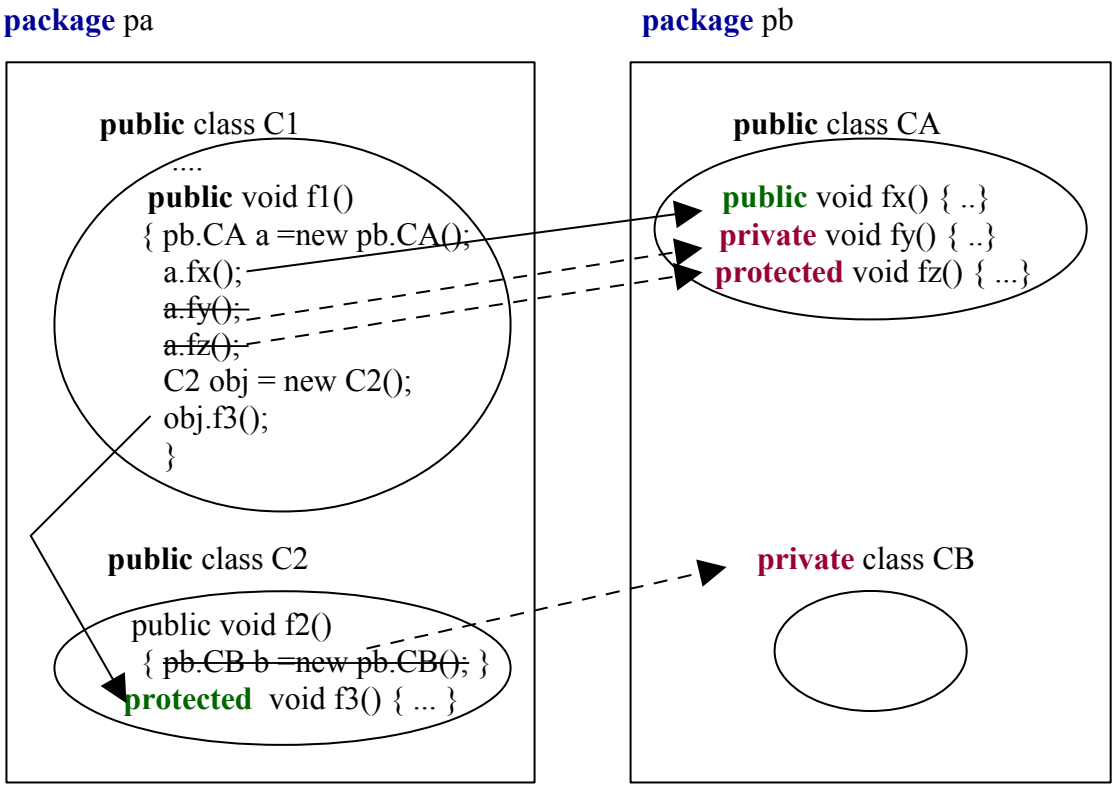
1.6. Résumé sur les modificateurs d'accès:

Accès depuis le package courant:

modificateur d'accès (**m**)	**héritage possible** d'une classe déclarée *m* pour une sous-classe du même package	membre déclaré *m* **accessible** depuis le code d'une autre classe du même package
par défaut (rien d'indiquer)	oui	oui
public	oui	oui
protected	oui	oui
private	**non**	**non**

Accès depuis un autre package:

modificateur d'accès (**m**)	**accès (ou héritage) possible** vers une classe déclarée *m* pour une classe d'un autre package	membre déclaré *m* **accessible** depuis le code d'une autre classe d'un autre package
par défaut (rien d'indiquer)	non	non
public	**oui**	**oui**
protected	**oui**	non (sauf si depuis sous-classe)
private	non	non



2. Gestion des exceptions

- Une **exception** est une espèce de **signal** qui indique qu'un **événement exceptionnel (erreur, condition non vérifiée) est advenu**. Les exceptions de JAVA seront en fait des ***instances des classes d'exception***.
- **Lancer ou lever une exception** consiste à **indiquer que quelque chose d'exceptionnel vient de se passer**.
- **Capturer une exception** consiste à indiquer que l'on va la **gérer** (essayer de rattraper le coup ou afficher un message d'erreur significatif).

Les exceptions se propagent en remontant les appels de fonctions qui ont été effectués.
Si une exception n'est pas gérée par le bloc qui l'a lancé, on dit que le bloc est négligeant.
L'exception se propage alors vers le bloc (ou fonction) directement englobant. Si aucun des niveaux ne gère l'exception, l'interpréteur JAVA arrête alors le programme après avoir affiché un message d'erreur.

Intérêt du mécanisme:

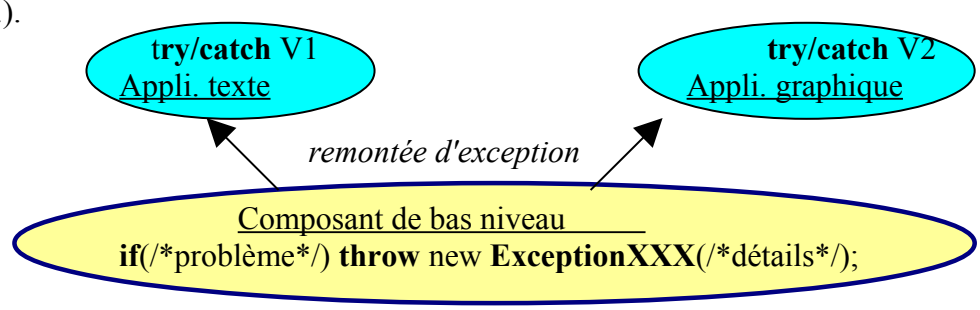
Lorsqu'une fonction de bas niveau s'aperçoit qu'il y a un problème (fichier non existant, connexion impossible, ...) elle ne connaît généralement pas le contexte global de l'application et ne sait pas s'il faut:

- Afficher un message d'erreur en mode texte (sur la console).
- Afficher un message en mode graphique dans une boîte de dialogue.
- Retourner l'erreur vers le client si l'erreur s'est produite coté serveur.
- Enregistrer le problème dans un fichier le log.
- ...

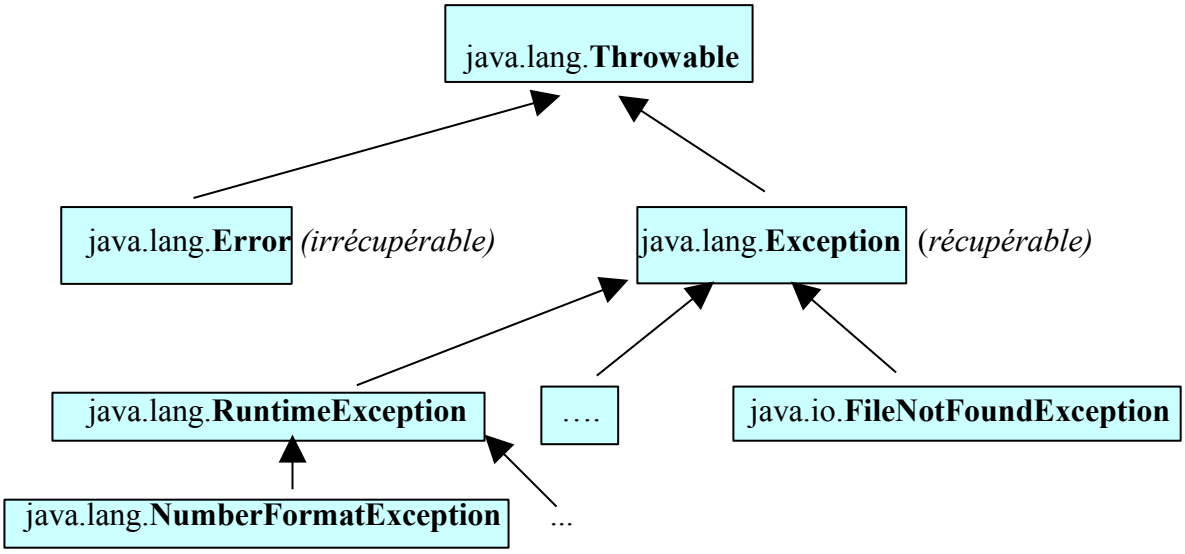
Tout traitement d'erreur en "dur" n'est donc jamais complètement satisfaisant dans tous les cas de figure et restreint donc le panel d'utilisation du composant de bas niveau.

L'autre solution qui consiste à retourner systématiquement un code d'erreur (éventuellement du genre ERROR_SUCCESS) n'est pas non plus très élégant car il oblige le code appelant à systématiquement tester (à coup de if/else ou switch) la valeur de retour de la fonction de bas niveau pour savoir si tout s'est bien passé.

Le mécanisme de traitement des exceptions apporte la meilleur solution car il permet de remonter simplement une indication d'exception de la couche N (de bas niveau) vers la couche N+1 (de haut niveau). Cette remontée d'exception n'est déclenchée qu'en cas de problème (rien ne se passe quand tout se passe bien).
La couche N+1 (qui reçoit l'exception) connaît généralement mieux le contexte global de l'application et va pouvoir déclencher un traitement approprié (en mode texte, en mode graphique, ...).



2.1. Classes d'exceptions:



Remarques:

- Tout type d'exception dérivant de **Exception** peut être rattrapé au moyen d'une bloc try/catch .
- Toute exception dont le type (ou sous type) n'est pas **RuntimeException** doit obligatoirement être gérée à un certain niveau pour que le code puisse être compilé. Par contre une exception dérivant de **RuntimeException** n'a pas à être systématiquement traitée (le try/catch est alors facultatif).

2.2. Levée (lancement) d'une exception:

Le mot clé **throw** (conceptuellement équivalent à un return) permet de quitter une fonction dans un cas d'erreur en lançant une exception qu'il faudra alors rattraper au dehors.

if(***/*problème*/*** **)**

```
throw  new MyException("paramètre , contexte de l'exception");
```

MyException peut être l'une des nombreuses classes prédéfinies d'exception de JAVA (EOFException, …) ou bien une nouvelle classe dérivant de java.lang.Exception .

2.3. Déclaration d'exceptions:

Une fonction (méthode) JAVA doit déclarer la liste des types d'exceptions qu'elle est susceptible de lever (ou de laisser remonter):

exemples:

public void open_file() **throws IOException** { … }

public void mafonction(param1,…) ***throws MyException1, MyException2*** {… }

NB:

- Une fonction qui ne gère pas en interne certaines exceptions est dite ***négligente*** et ne pourra être compilée que si les exceptions non gérées à ce niveau sont déclarées après le mot clef **throws**.
- La partie **throws** d'un prototype de méthode permet d'***indiquer à un futur (autre) programmeur*** quelles sont ***les exceptions qui pourront éventuellement être déclenchées*** lorsque ce code de bas niveau sera utilisé. Cette information est très utile pour savoir quels sont les "try/catch" nécessaires.

2.4. Gestion d'exceptions:

Les combinaisons possibles sont **try/finally** ou **try/catch** ou **try/catch/finally**

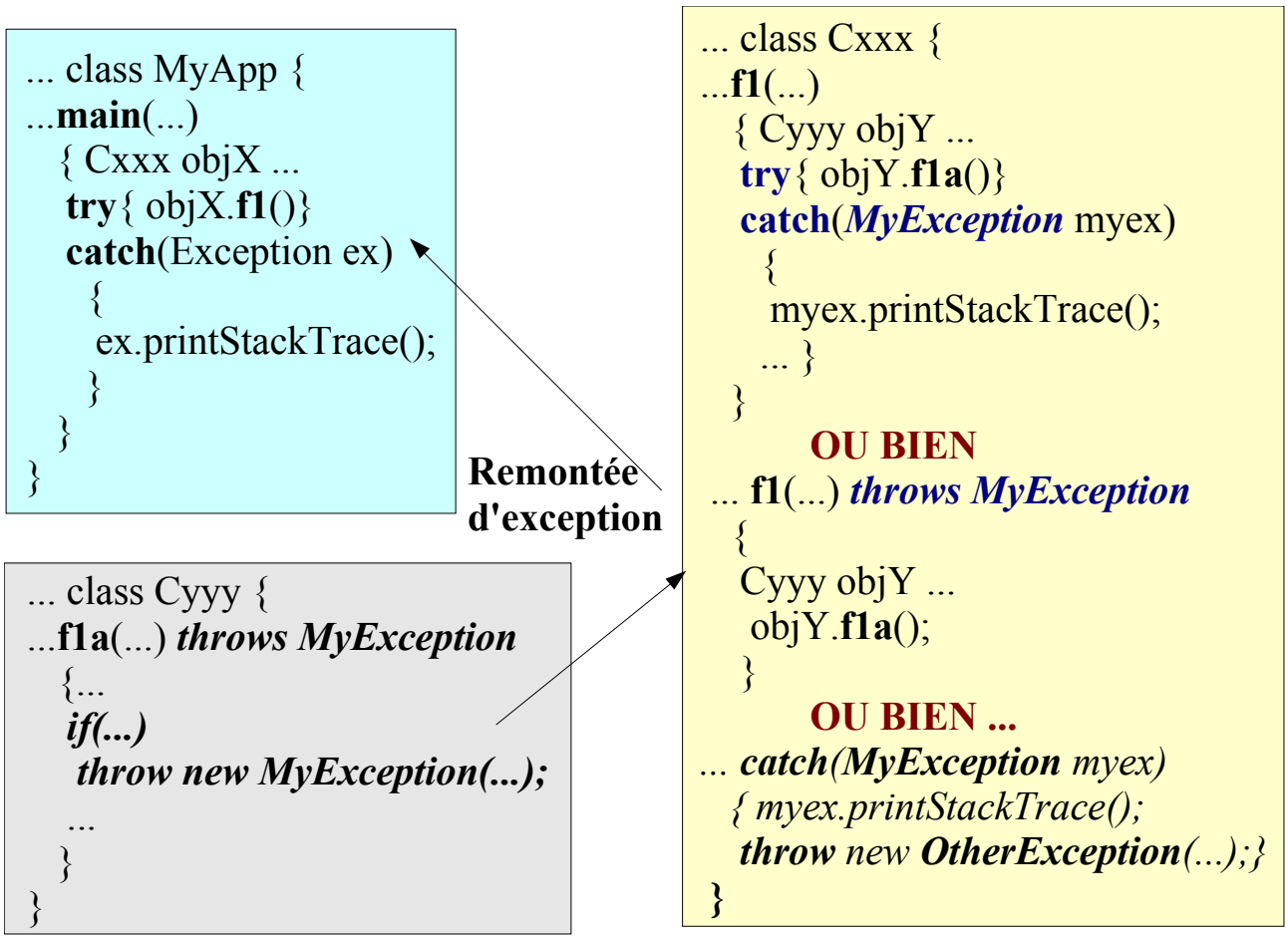
```
try {
… // Instructions sous contrôle. Comportement normal si tout se passe bien.
… // On sort définitivement de ce bloc en cas d'exception.
… // Si on tombe sur une instruction de type return on exécute alors le bloc finally
… // avant de sortir.
}

catch(TypeException1  e1)
{
… // Gestion de l'exception e1 qui est une instance de TypeException1
… // ou d'une sous classe.
}
catch(TypeException2 e2)
{
System.err.print("Exception: " + e2.getMessage());
}
catch(Exception ex  /* n'importe quelle autre exception */ )
{
ex.printStackTrace(); // très pratique pour le debug
}

finally
{
… // Code exécuté dans tous les cas de figure (après fin normal des instructions,
… // après exception traitée ou pas, après un return du bloc try ).
}
```

NB: En cas d'exception , il est souvent utile de générer des lignes au sein de certains fichiers de logs.

NB2: **Depuis le jdk 1.7** il est possible d'écrire

catch(TypeException1 e1, TypeException2 e2) { ..... }

2.5. Niveau intermédiaire – alternatives de traitement (exceptions)



Le niveau intermédiaire peut au choix:

- **traiter lui même l'exception** (via try/catch)
- **laisser remonter l'exception telle quelle sans la gérer** (sans try/catch mais throws ....)
- **traiter l'exception et en propager une nouvelle version** (throw new OtherException ...)

NB: La troisième solution est la plus appropriée dans un environnement n-tiers , car chaque niveau intermédiaire peut alors :

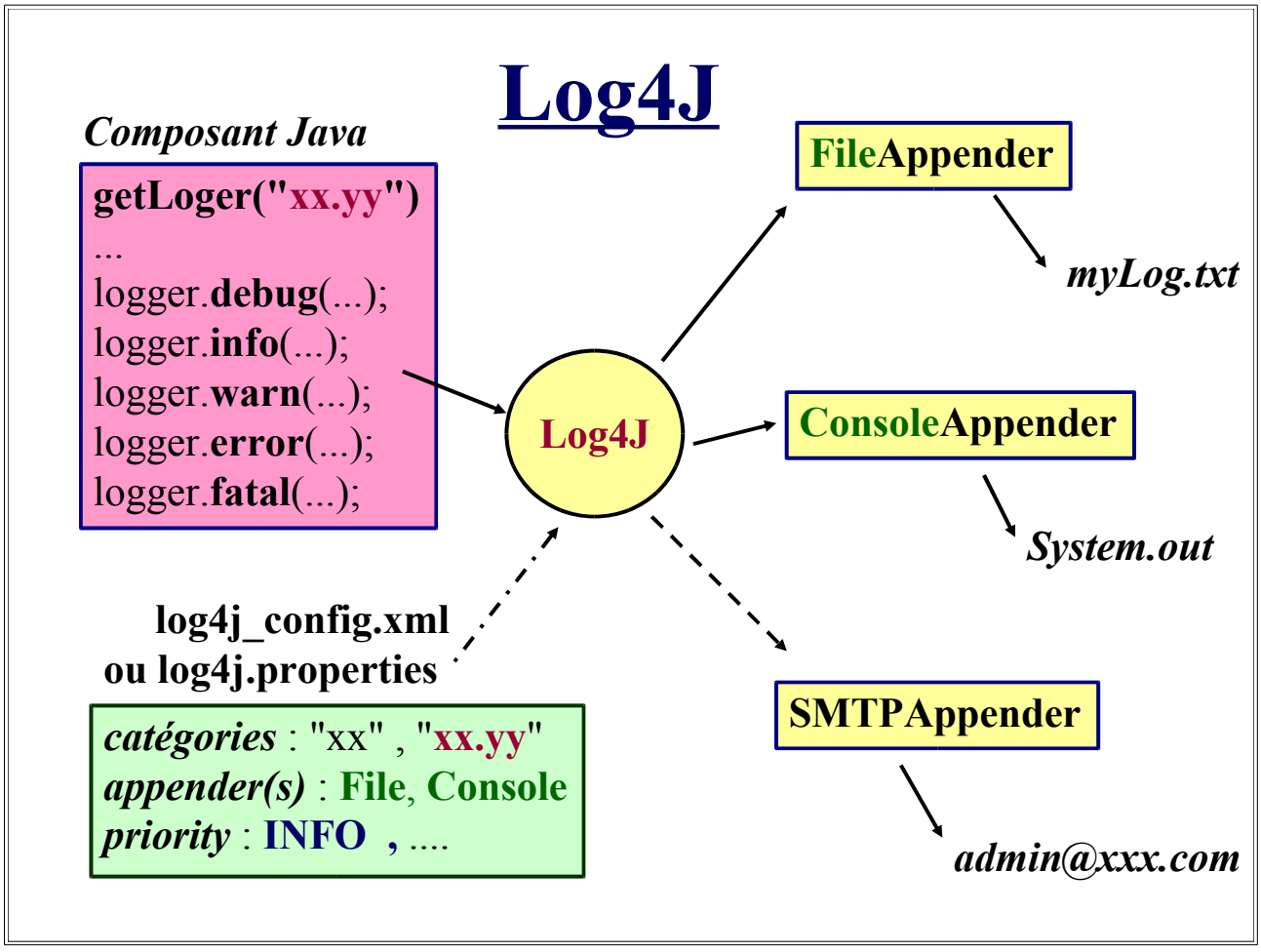
- générer des logs sur le problème vu localement (avec détails techniques)
- remonter une exception dans le format attendu par le niveau appelant

3. Présentation des API de log

3.1. Différentes API disponibles

- Historiquement , **Log4J** fut la première API de log efficace dans le mode Java .
- Le **JDK 1.4** a par la suite introduit un équivalent se voulant standard => *package* **java.util.logging**
- La communauté open source "***Apache***" a ensuite proposé une petite API chapeautant les 2 premières. L' API **org.apache.commons.logging** utilise en interne Log4J si présent ou bien java.util.logging du JDK1.4 sinon.
- Encore quelques années plus tard, est apparue une nouvelle Api "**slf4j**" (simple log facade for java) qui (comme commons-logging) est également une petite API de haut niveau délégant à des sous API (ex: log4j , ...) . **slf4j** impose cependant un choix statique dès la construction de l'application et se montre plus fiable et plus performante que commons-logging. Pour faire "comme tous le monde" à partir de 2010 , il est grandement conseillé d'utiliser "**slf4j**" . Slf4j est par exemple utilisé en interne par les versions récentes de Spring et d'Hibernate .

3.2. Concepts communs



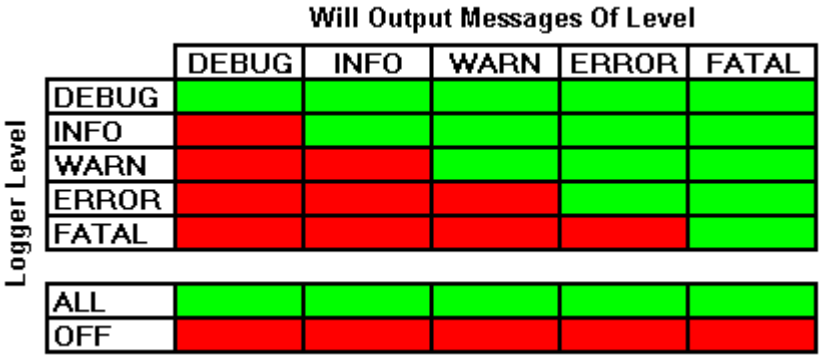
Via l'une de ces API , le développeur génère des logs (avec message et niveau de gravité) sans avoir

à connaître le fichier de destination ni les règles de filtrage.

Via un fichier de paramétrage (.xml ou .properties) , un administrateur peu effectuer quelques réglages fins:

- Niveau des lignes de logs à récupérer (ERROR, WARNING , INFO , ...)
- Destination(s) (fichier texte , fichier xml, console , ...)

Principal avantage ==> pas de if(DEBUG) dans le code et donc pas de perte de temps CPU pour tenir compte des paramétrages.



3.3. Génération des lignes de logs (slf4j)

NB: ***slf4j-api-1.5.6.jar*** , ***slf4j-log4j12-1.5.6.jar*** et ***log4j-1.2.15*** doivent être ajoutés au "CLASSPATH"

```
package tp;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Essai_Slf4j {
	private /*static*/ Logger logger = LoggerFactory.getLogger(Essai_Slf4j.class);

	public void doJob()
	{
	int x=1;		int y=0;
			try  {
				int z=x/y;
			}
			catch(Exception ex) {
				logger.error("error msg",ex);  // logger.fatal("Fatal Error");
			}
		logger.warn("my warning");
		logger.info("my info");
		logger.debug("my debug");
		//logger.trace("my trace"); ?
	}
}
```

...

3.4. Configurations de la sortie des logs (via slf4j et log4j)

log4j.properties

```
log4j.rootLogger=debug, CONSOLE , RFILE

log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
# Pattern to output the caller's file name and line number.
log4j.appender.CONSOLE.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n

log4j.appender.RFILE=org.apache.log4j.RollingFileAppender
log4j.appender.RFILE.File=mylog.txt
log4j.appender.RFILE.MaxFileSize=100KB
log4j.appender.RFILE.MaxBackupIndex=1
log4j.appender.RFILE.layout=org.apache.log4j.PatternLayout
log4j.appender.RFILE.layout.ConversionPattern=%d %p %t %c - %m%n
```

4. Propriétés du système

JAVA ne supporte pas directement les **variables d'environnement** du système car il est censé être indépendant de la plate-forme. **JAVA** propose en remplacement la notion de "**liste de propriété du système**":

Récupération d'une propriété au sein du programme:

```
String ModeDebug = System.getProperty("monAppli.Debug");
```

L'option **–D** de l'interpréteur JAVA permet de fixer une propriété sur la ligne de commande:

```
java –DmonAppli.Debug=true  packagexxx.monAppli
```

Quelques propriétés prédéfinies:

user.home	home directory
java.home	répertoire d'installation de java
java.class.path	path pour trouver et charger les fichiers .class
os.name	nom du système d'exploitation hôte
line.separtor	ex: "\n" sous Unix
path.separator	Ex: ":" sous Unix , ";" sous PC
user.name	nom de l'utilisateur courant
user.dir	répertoire courant de l'utilisateur

Arguments de la ligne de commande et valeur en retour :

```
public class MyEcho {
        public static void main(String argv[])
                {
                int argc = argv.length;  // Nb: argv[0] correspond au premier argument
                for(int i=0;i<argc;i++)
                        System.out.print(argv[i]+" ");
                if(...)
                        System.exit(0); // code de retour dont la signification dépend de l'OS.
                ... }
}
```

Lignes typiques de commandes à placer dans un fichier *.bat* :

```
set PATH=%PATH%;C:\prog\JAVA\j2sdk1.4.2_06\bin

set CLASSPATH=%CLASSPATH%;c:\repx\xxx.jar;c:\repy

java -classpath c:\repz\autreApi.jar  -Dprop1=val1 -Dprop2=val2  package_p.MonAppli arg1
arg2  .... argn     > ficRes.txt     2> ficErr.txt
```

5. Quelques structures de données (java.util)

NB: **Les classes suivantes sont disponibles dans toutes les versions du JDK (1.0 , 1.1 , ...)**

5.1. Liste indexée d'objets – classe Vector

```
Vector liste = new Vector();
liste.addElement( obj1 );
liste.addElement( new Integer(5) ); // on ne peut ajouter que des objets (pas des "int").
liste.addElement( obj3 );  // depuis le jdk 1.2 la nouvelle méthode add() fait la même chose
liste.addElement( new String("abc") );
…
System.out.println("nombre d'éléments= " + liste.size() );

for (int i=0; i < nbElements ; i++)
        {  Object obj = liste.elementAt(i);
          /* if(obj instanceof  String) */  // test à effectuer si on est pas sûr du type
              chExt = ( (String) obj).substring(n,n-3);   }
index = liste.indexOf(obj2);
liste.removeElementAt(index);
liste.removeElement(obj1);

obj = liste.firstElement();
obj = liste.lastElement();

liste.setElementAt(obj,index);

liste.removeAllElements();
if (liste.isEmpty() ) …
```

5.2. Pile d'objets – classe Stack

```
Stack pile = new Stack(); // sous classe de Vector
pile.push(obj1);    pile.push(obj2);
try {
        lookedObj = pile.peek(); // récupère l'objet en sommet de pile sans l'enlever.
        obj = pile.pop();
    }
catch (EmptyStackException e) { …}
…
if (pile.empty()) …
```

5.3. Table de hachage - classe HashTable

[Equivalent plus récent conseillé à partir du jdk1.2 : ***HashMap*** *]*

La classe concrète **HashTable** (héritant de la classe abstraite **Dictionary**) permet de stocker des objets et d'y accéder par des clés.

Une clé est un objet d'une classe quelconque héritant de Object.

La classe racine ***java.lang.Object*** contient à cet effet une méthode appelée ***hashCode***() permettant de calculer la valeur de la clé interne du mécanisme. Cette méthode peut éventuellement être redéfinie dans les sous classes.

L'exemple le plus classique de clé est une chaîne de caractères (classe String).

Exemple:

```
HashTable  tableObjets = new HashTable();
tableObjets.put("NomObj1", obj1);
tableObjets.put("NomObj2", obj2);
…
tableObjets.put("NomObjN", objN);

MaClasse obj = (MaClasse) tableObjets.get("NomObjetRecherché");
if (obj != null) …

tableObjets.remove("NomObj2");

NbElts = tableObjets.size();

if(tableObjets.contains(obj1)) ….
if(tableObjets.containsKey("NomObj1")) …


Enumeration  parcoursElts = tableObjets.elements();
Enumeration parcoursCles = tableObjets.keys();

while (parcoursElts.hasMoreElements())
{
obj = (MaClasse) parcoursElts.nextElement();
…
}

if( ! tableObjets.isEmpty())
tableObjets.clear();
```

NB: Une sous classe de HashTable appelée **Properties** permet stocker ou lire les associations (clé,valeur) au niveau d'un flux (stream).

6. Collections (depuis le jdk 1.2)

Implémentations → / *Interfaces ↓*	*Hash Table*	*Resizable Array*	*Balanced Tree*	*Linked List*	*classes du jdk1.1 et réadaptées sur les collections du jdk1.2*
Set	**HashSet**		**TreeSet**		
List		**ArrayList**		**LinkedList**	**Vector**
Map	**HashMap**		**TreeMap**		**Hashtable**

NB: Les interfaces **Set**, **List** (et des parties internes de **Map**) héritent de l'**interface Collection**

Set ==> Ensemble d'éléments (pas de duplication possible)
List ==> Liste ordonnées d'éléments (doublons éventuels) + Accès via index
Map ==> Table d'association [ensemble de couples (clef,valeur)]

NB: Les éléments internes des **TreeSet** et **TreeMap** sont **ordonnés**.
----> Les éléments internes des **HashSet** et **HashMap** ne sont **pas ordonnés** (*ordre variable au cours du temps suite à des ajouts ou suppressions*) .

6.1. Interface Collection

principales méthodes	***détails***
boolean **add**(Object o)	retourne false si l'élément y était déjà ou si ...
boolean **addAll**(Collection c)	
void **clear**()	vide tout
boolean **contains**(Object o)	objet contenu dans la collection ?
boolean **isEmpty**()	
Iterator **iterator**()	retourne un itérateur pour parcourir la collection
boolean **remove**(Object o)	
int **size**()	Nombre d'éléments de la collection
Object[] **toArray**()	
...	

Nb: un **itérateur** s'utilise de la façon suivante:

```
Iterator  it = MaCollection.iterator();
while(it.hasNext())
    {
     Object obj  = it.next();
    obj. ....
    }
```

NB: Depuis le jdk 1.2 , il est conseillé d'utiliser **Iterator** à la place de ~~**Enumeration**~~.

NB: La classe **Vector** est *synchronisée* (accès concurrents possibles depuis différents threads).
----> **ArrayList** et les **autres collections** ne sont **pas synchronisés**.
Cependant la méthode statique ***Collections.synchronizedList(liste)*** permet d'obtenir une version synchronisée lorsque c'est nécessaire .

L'obtention d'un itérateur et le parcours associé doit quelquefois être effectué au sein d'un bloc {...} encadré par *synchronized(liste)* de façon à garantir une cohérence au niveau de l'itérateur lors d'éventuels accès concurrents (plusieurs threads).

Exemple:

```
List list = Collections.synchronizedList(new ArrayList());
   ...
synchronized(list) {
   Iterator i = list.iterator(); // Must be in synchronized block
   while (i.hasNext())
                f1(i.next());
}
```

6.2. Interface List

L'interface **List** hérite de l'interface **Collection** et offre en plus les méthodes suivantes:

principales méthodes	***détails***
Object **get**(int index)	retourne l'objet à la position indiquée
int **indexOf**(Object o)	retourne -1 si l'objet de fait pas partie de la liste
int **lastIndexOf**(Object o)	idem pour dernière occurence trouvée
ListIterator **listIterator**()	
Object **set**(int index,Object o)	remplace l'élément en position index
List **subList**(int first, int last)	du premier index inclus au dernier exclus

NB:
L'interface **ListIterator** hérite de Iterator et offre en plus :

- un parcours dans le sens inverse (via **HasPrevious**() et **previous**())
- un parcours basé sur les index (via **nextIndex**() et **previousIndex**()).

6.3. Interface Map

L'interface **Map** représente les fonctionnalités d'une **association** (ensemble de couple Key/Value)

principales méthodes	*détails*
boolean **containsKey**(Object key)	
boolean **containsValue**(Object v)	
void **clear**()	vide tout
Object **get**(Object key)	
boolean **isEmpty**()	
Object **put**(Object key,Object val)	
boolean **remove**(Object o)	
int **size**()	Nombre d'éléments de la collection
Set **entrySet**()	vue sous la forme d'un ensemble d'éléments de type Map.Entry ==> (get/set Key/Value ())
Collection **values**()	
...	

6.4. Tri

La classe **Collections** (*avec un **s** à la fin*) comporte une méthode statique ***sort*** permettant de trier les éléments de la liste passée en paramètre:

```
java.util.Collections.sort(liste);
```

NB: L'ordre de tri est par défaut piloté par l'interface **java.lang.Comparable** implémentée par des éléments tels que ceux des types *Integer , String , Double , ...*

NB: Un vecteur étant considéré comme une liste (depuis le jdk 1.2), on peut le trier via le mécanisme précédent.

NB: une seconde version de **Collections.sort()** admet un second paramètre de type **java.util.Comparator** de façon à contrôler finement l'ordre de tri (lorsque c'est nécessaire).

L'interface **Comparator** nous impose de programmer une méthode

```
public int compare(...o1,...o2){
    //si o1 est plus petit que o2 alors return -1 (ou valeur négative);
    //si o1 est égal à o2 alors return 0;
    //si o1 est plus grand que o2 alors return +1 (ou valeur positive);
}
```

7. Generics (depuis Java 5)

7.1. Incohérence de type sur les éléments d'une collection

Soit une classe utilisant une Collection d'objet String et parcourue grâce à un itérateur :

```
import java.util.*;

public class Ex1
{

  private void testCollection() {
    List list = new ArrayList();
    list.add(new String("Hello world!"));
    list.add(new String("Good bye!"));
    list.add(new Integer(95));          // incohérence
    printCollection(list);              // pas détectée
  }                                     // à la compilation

  private void printCollection(Collection c) {
    Iterator i = c.iterator();
    while(i.hasNext()) {
      String item = (String) i.next();
      System.out.println("Item: "+item);
    }
  }

  public static void main(String argv[]) {
    Ex1 e = new Ex1();    e.testCollection();
  }
}
```

Bien que comportant une incohérence (le troisième élément de la *Collection* est un *Integer)*, ce code passe bien à la compilation .
Le problème vient du fait qu'il est nécessaire de faire un CAST sur le résultat de la méthode *i.**next()***.
Seule l'exécution de ce programme provoquera une **ClassCastException.**

7.2. Classes génériques fortement typées

Les classes génériques java permettent d'utiliser une collection avec un type donné et de vérifier les erreurs de Cast. En compilant l'exemple ci-après avec l'option *–source 1.5*, on obtient une erreur lorsque l'on essaye d'ajouter un objet *Integer* dans notre *Collection* de *String*.

```
import java.util.*;

public class Ex2 {

  private void testCollection() {
    List<String> list = new ArrayList<String>();
    list.add(new String("Hello world!"));
    list.add(new String("Good bye!"));
    list.add(new Integer(95)); // ERREUR
    printCollection(list);
  }

  private void printCollection(Collection<String> c) {
    Iterator<String> i = c.iterator();
    while(i.hasNext()) {
      String item = i.next();       // plus de cast à expliciter
      System.out.println("Item: "+item);
    }
  }

  public static void main(String argv[]) {
    Ex2 e = new Ex2();
    e.testCollection();
  }
}
```

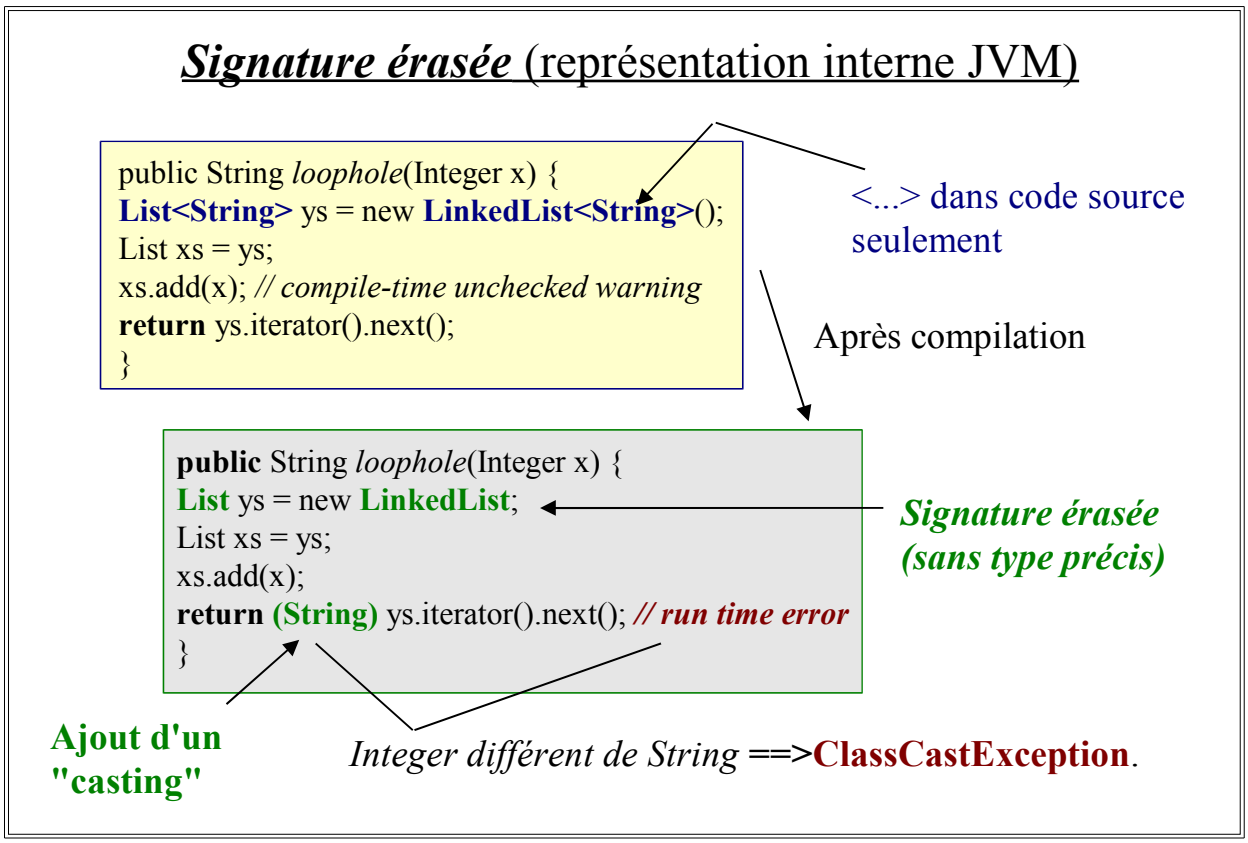
Prédéfinis (en mieux) dans java.util:

```
public interface List<E> {
        void add(E x);
        Iterator<E> iterator();
       ...
}

public interface Iterator<E> {
        E next();
        boolean hasNext();
       ...
}
```

7.3. Java Generics vs. C++ Templates

Même si les classes génériques java ressemblent aux templates C++, elles sont différentes. Les classes "Generics" JAVA font simplement une **vérification** de type à la compilation et éliminent l'utilisation explicite des CAST.

Le C++ construit quant à lui différentes classes "assez statiques (ex: " vector<int> , vector<double> , ...) à partir d'un unique modèle générique de code appelé "template" (ex: Vector<T>) .



Wildcards de java >=5 .

Wildcards (?) de Java 5

? = wildcard (any type)
Collection<?> = Collection of Unknown .

```
void printCollection(Collection<?> c) {
for (Object e : c) {
   System.out.println(e);
}}
```

```
public void drawAll(List<? extends Shape> shapes) {
// liste de chose héritant de Shape --> appel possible avec List<Circle> ou ...
    for (Shape s: shapes) {s.draw(this);  // ok : opération en lecture/affichage
}}

public void addRectangle(List<? extends Shape> shapes) {
   shapes.add(0, new Rectangle()); // compile-time error!
   // même si Rectangle hérite de Shape , car opération impossible
  // si ? Remplace Circle héritant de shape
}
```

On peut programmer de nouvelle classes génériques comme le montre l'exemple suivant:

```
package tp;

import java.util.ArrayList;
import java.util.List;

/* LIFO = Last In , First Out ,
    mieux que FINO = First In, Never Out*/

public class MyStack<T> {
     private List<T> listeInterne = new ArrayList<T>();

     public void push(T objVal){
          listeInterne.add(objVal);
     }

     public T pop(){
          T objVal = null;
          if(!listeInterne.isEmpty()){
               int lastIndex = listeInterne.size() - 1;
               objVal = listeInterne.remove(lastIndex);
          }
          return objVal;
     }
}
```

test/utilisation:

```
MyStack<Integer> pileEntiers = new MyStack<Integer>();
     pileEntiers.push(2); pileEntiers.push(1);
     System.out.println(pileEntiers.pop());
     System.out.println(pileEntiers.pop());
```

```
MyStack<String> pileChaines = new MyStack<String>();
     pileChaines.push("abc"); pileChaines.push("def");
     System.out.println(pileChaines.pop());
     System.out.println(pileChaines.pop());
```

7.4. Nouvelle boucle for (sémantique "for each")

La nouvelle version de la boucle ***for*** permet de parcourir des collections et des tableaux sans utiliser explicitement d'itérateur ni d'index.

Nouvelle syntaxe :

```
for (FormalParameter : Expression) Statement
```

L'expression doit être un tableau ou l'instance d'une nouvelle interface *java.lang.**Iterable***, nécessaire pour l'utilisation de la nouvelle boucle.
L'interface ***java.util.Collection*** **implémente dorénavant** ***Iterable***.

Exemple avec une collection :

```
public void oldFor(Collection c) {
  for(Iterator i = c.iterator(); i.hasNext(); ) {
    String str = (String) i.next();
    System.out.println(str);
  } }
```

Peut être ré-écris avec içi une utilisation conjointe d'une classe générique java :

```
public void newFor(Collection<String> c) {
  for(String str : c) {
      System.out.println(str);
  } }
```

Exemple avec un tableau :

```
public int sumArray(int array[]) {
  int sum = 0;
  for(int i=0;i<array.length;i++) {
    sum += array[i];
  }
  return sum; }
```

peut être ré-écris:

```
public int sumArray(int array[]) {
  int sum = 0;
  for(int i : array) {
    sum += i;
  }
  return sum; }
```

Cette nouvelle version de la boucle for ne remplacera pas toujours l'ancienne. L'index i étant parfois nécessaire dans le traitement.
NB: La nouvelle boucle **for** de **JAVA 5** ressemble fortement à la boucle ***foreach*** de **C#**.
La nouvelle boucle aurait pu s'écrire **:** `foreach(int i in array)`
Sun a préféré ne pas créer de nouveau mots clés par soucis de compatibilité.

V - Classes utilitaires , aspects divers

1. Eléments de Java >=5 (jdk 1.5 , 1.6 et 1.7)

1.1. Enumérations

Il est maintenant possible de créer des énumérations comme en C/C++.
Exemple de déclaration :

```
public class Cxx {
public enum Jour { DIMANCHE , LUNDI, MARDI, ..., SAMEDI };

private Jour jour = Jour.DIMANCHE; // valeur par défaut ici.

public Jour getJour(){ return this.jour; }
public void setJour(Jour jour} { this.jour = jour; }
}
```

Le mot clef **enum** est maintenant réservé. Son utilisation nécessite l'option de compilation `-source 1.5` (ou 1.6 , 1.7) du compilateur javac .

- **Jour** est ainsi considéré comme un **type** de données . Il s'agit d'une classe implémentant implicitement les interfaces **Serializable** et **Comparable**.
  Cette classe Java d'énumération héritant implicitement de **Object** surcharge de façon adéquat les méthodes *toString*() , *equals*() et *hashCode*() .
- Les éléments internes de l'énumération sont codés comme des constantes entières statiques (valeurs = 0 ,1 , ...) ==> méthode **ordinal()**
- Toute classe d'énumération (comme içi Jour) comporte une méthode statique **.values()** retournant le tableau des valeurs énumérables ainsi qu'une méthode **valueOf** (String name) retournant la valeur d'une constante selon son nom sous forme de chaîne de caractères.

exemple:

```
for (Jour j : Jour.values() )
        System.out.println(j + " [" + j.ordinal() + "]" );
```

```
....
objX.setJour(Jour.LUNDI);

Jour j = objX.getJour();

   switch(j){
   //NB: bizarrement pas de case Jour.LUNDI   mais case LUNDI
   case LUNDI:   System.out.println("Lundi c'est ravioli"); break;
   case MARDI:   System.out.println("Mardi c'est brocoli"); break;
   ...
   }
```

1.2. Fonctions à nombre d'arguments variable

```
void argtest(Object ... args) {
    for (int i=0;i <args.length; i++) {
        System.out.println(args[i]);
    }
  }

// invocation de la méthode
argtest("test", "data");
```

1.3. Sorties formatées (printf)

Les fonctions à nombres d'arguments variables ont permis l'implémentation de ***printf*** (identique à celle du langage C).

```
System.out.printf("%s %3d", name, age);
```

%s ==> string
%d ==> entier décimal (base 10)
%f ==> floating point
%g ==> floating point (quelque soit la précision : double ou float)

... voir l'aide en ligne sur *printf* pour approfondir la syntaxe.

1.4. import static (depuis Java 5)

```
import static  packagex.interfaceX.* ;
```

permet d'utiliser tous les éléments statiques de l'interfaceX (constantes , méthodes statiques , ...) sans avoir à les préfixer :

On peut alors se contenter d'écrire *CONSTANTE1* au lieu de *InterfaceX.CONSTANTE1* .

Exemples:

```
import static java.lang.Math.*;
import static java.lang.System.*;
```

- on peut écrire **PI** au lieu de ***Math.PI***
- **sin(x)** à la place de ***Math.sin(x)***
- **out.println("ok")** à la place de ***System.out.println("ok")***

2. Classes imbriquées (depuis jdk 1.1)

Les classes imbriquées *(Inner classes)* ont été introduites dans le JDK 1.1 pour pouvoir gérer relativement facilement les événements (voir chapitre AWT).

Exemples commentés:

Ce premier exemple montre que le niveau imbriqué peut directement accéder aux membres du niveau englobant:

```
public class ClasseContenante {
  public int x=1;
  private int y=2; // attribut privé de la classe de premier niveau

  private class ClasseImbrique {
    public int a=10;
    public int x=100; // attribut de la classe imbriquée

    public void aff()
        {    System.out.println("x du niveau imbrique : " + x);
                //this correspond içi a l'objet de la classe imbriqué:
                  System.out.println("this.a = " + this.a);
        //Le niveau imbriqué a directement accès aux membres de la
        //classe englobante (même s'ils y sont déclarés privés)
                  System.out.println("y = " + y);
        affContenant(); // appel direct sur le niveau englobant
        }
  }// fin classe imbriquée

  private void affContenant()  {
  System.out.println("---> fonction de la classe englobante :");
  System.out.println("x du niveau englobant : " + x);
  }

  public void essai()  {  ClasseImbrique obj = new  ClasseImbrique();
                          obj.aff();    }
}
```

Dans ce second exemple très classique , **la classe imbriquée n'a pas de nom** et l'indication **java.awt.event.ActionListener()** qui **précède le bloc entre { }** désigne içi le fait que la classe imbriquée **(hérite ou) implémente** l'interface **ActionListener** du package java.awt.event :

```
 ....
jButtonOK.addActionListener ( new
    /* début du code de la classe imbriquée (et anonyme) implémentant */
            java.awt.event.ActionListener()
   {
     public void actionPerformed(ActionEvent e)
     {
       jButtonOK_actionPerformed(e);
     }
   /* fin du code de la classe imbriquée */
   } );
 ....
```

===> classes de type ***XXX$1.class*** , ***XXX$2.class*** ... , une fois compilées.

3. Quelques classes utilitaires

3.1. Générateur de nombre aléatoire – classe Random

```
Random generateur = new Random( ); //initialisé avec le temps courant par défaut.
entierAleatoire = generateur.nextInt();
reelAleatoire = generateur.nextDouble();  // de 0.0 à 0.1
…
```

3.2. Gestion des instants (TimeStamp) – classe Date

```
Date d = new Date();              // Constructeur par défaut ➔Date d'aujourd'hui.
Date d2 = new Date(nbMsEcouleesDepuis01_01_1970);
//Date d3 = new Date(annee,mois,jour); /*deprecated */
//Date d4 = new Date(annee,mois,jour,heure,min,sec); /*deprecated */

long nbMsEcouleesDepuis01_01_1970 = d.getTime();
d.setTime(nbMsEcouleesDepuis01_01_1970 );

System.out.println("date = " + d.toString() ");
System.out.println("date GMT = " + d.toGMTString() "); /*deprecated */
System.out.println("jour du mois = " + d.getDay() ");/*deprecated */      // getXX() ➔➔ setXX(x)
System.out.println("mois = " + d.getMonth() "); // de 0 à 11 /*deprecated */
System.out.println("année = " + d.getYear() ");  /*deprecated */
System.out.println("heure=" + d.getHours() + "/" + d.getMinutes() + "/"  + d.getSeconds());/*deprecated */
```

Remarque: une grande partie des méthodes de la classe Date est aujourd'hui obsolète (pas d'internationalisation, problème sur le calendrier). Il est donc fortement conseillé d'utiliser conjointement les nouvelles classe java.util.**Calendar** et java.text.**DateFormat**.

La classe **Date** devrait normalement n'être utilisée que pour stocker une valeur de type "Date + Heure (TimeStamp)" (exprimée en **ms**).

3.3. gestion des dates du calendrier - classe Calendar

Calendar cal = **Calendar.getInstance()**; *//pas de new (Calendar = classe abstraite)*
// la méthode de classe getInstance() retourne généralement une instance de la sous classe
// concrète GregorianCalendar.

L'instance créée par **Calendar.getInstance**() comporte la date d'aujourd'hui et l'heure actuelle comme valeurs par défaut.
Pour changer certaines valeurs, il faut utiliser une des méthodes **set**(...).

```
cal.setTime(date /* de type Date et comportant nbMsEcouleesDepuis01_01_1970 */ )
```

cal.**set**(int ***annee***,int ***mois***,int ***jourDuMois***);
cal.**set**(int ***annee***,int ***mois***,int ***jourDuMois***,int ***heure***,int ***minute***,int ***second***);
Remarque importante: chaque variante d'utilisation de la méthode **get**(int field) ci-dessous est associée à une variante **set**(int field,int value) pour fixer la valeur du champ.

int seconde = cal.**get**(Calendar.**SECOND**); *// de 0 à 59*

int minute = cal.**get**(Calendar.**MINUTE**); *// de 0 à 59*

int heure = cal.**get**(Calendar.**HOUR_OF_DAY**); *// de 0 à 23*

int jourDuMois = cal.**get**(Calendar.**DAY_OF_MONTH**); *// de 1 à 31*

int mois = cal.**get**(Calendar.**MONTH**);
*// Le numéro du mois va de **0** (Calendar.JANUARY) à **11** (Calendar.DECEMBER)*

int annee = cal.**get**(Calendar.**YEAR**);

int jourDeLaSemaine = cal.**get**(Calendar.**DAY_OF_WEEK**);
// le jour de semaine va de 1(Calendar.SUNDAY) à 7(Calendar.SATURDAY)

int semaineDansAnnee = cal.**get**(Calendar.**WEEK_OF_YEAR**); *// de 1 à 52*

```
Date date /*instant*/ = cal.getTime();
```

Remarque:
Pour afficher une date sous forme de chaîne de caractères , il faut utiliser conjointement la classe **DateFormat** du package **java.text** (voir paragraphe suivant)

4. Internationalisation

La classe **Locale** permet la récupération automatique des éléments spécifiques à l'aspect "régional" du poste utilisateur (langue, symbole monétaire , ...).
Cette classe est bien souvent automatiquement utilisée par beaucoup d'autres classes (exposées ci-après):

Un programme devant fonctionner dans différentes langues devrait récupérer les libellés de la façon suivante:

```
ResourceBundle myResources =  null;
myResources = ResourceBundle.getBundle("mypackage.MyResources");
String libelle = myResources.getString("msg.welcome");
```

La méthode statique **ResourceBundle.getBundle**("*mypackage.*MyResources") va rechercher une classe vérifiant les points suivants:

- de nom *mypackage.*MyResources_xx où xx est le code du pays local (*ex:* MyResources**_fr ,** MyResources**_de , ...**)
- héritant de **ResourceBundle** (ou d'une de ses sous classes **ListResourceBundle** , ...).

version par défaut (utilisée si `MyResources_fr n'est pas trouvée`):

```
public class MyResources extends java.util.ListResourceBundle {
      public Object[][] getContents() {
               return contents;
      }
      static final Object[][] contents = {
               {"msg.welcome", "welcome"},
               {"msg.goodbye", "goodbye"},
      };
}
```

version francaise:

```
public class MyResources_fr extends java.util.ListResourceBundle {
      public Object[][] getContents() {
               return contents;
      }
      static final Object[][] contents = {
               {"msg.welcome", "bienvenue"},
               {"msg.goodbye", "au revoir"},
      };
}
```

NB: Si la classe *mypackage.*MyResources n'est pas trouvée dans le CLASSPATH , la méthode **ResourceBundle.getBundle**("*mypackage.MyResources*") recherche alors un **fichier texte** de nom **MyRessources.properties** *(ou **MyRessources_fr.properties** ou ...)*.

Ces fichiers "***.properties***" sont également à placer au sein du sous répertoire "mypackage" situé au niveau du CLASSPATH.

Exemple: ***MyResources.properties***

```
msg.welcome=welcome
msg.goodbye=goodbye
```

MyResources_fr.properties

```
msg.welcome=bienvenue
msg.goodbye=au revoir
```

5. Mise en forme du texte (java.text)

Le package **java.text** comporte des classes permettant de mettre en forme textuelle des choses diverses (nombres , dates , ...) .

5.1. DateFormat

```
DateFormat df = DateFormat.getDateInstance(DateFormat.LONG);
String chDate = df.format(date);
Date d  = df.parse(chaineDate);

DateFormat tf = DateFormat.getTimeInstance(DateFormat.LONG);
String chHeure = tf.format(date);

DateFormat dtf = DateFormat.getInstance(DateFormat.LONG);
String chDateEtHeure = dtf.format(date);
```

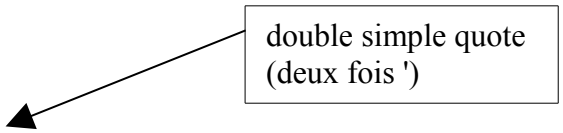
5.2. MessageFormat

La classe MessageFormat comporte une fonction statique essentielle dénommée format qui permet de générer une chaîne de caractères à partir d'un format générique au sein duquel certains éléments spéciaux ({0} , {1}, {2}, ...) sont automatiquement remplacés par les éléments d'un tableau de valeurs passé en paramètre. Chaque élément du tableau doit être de type Object et comporte donc une méthode toString() qui va bien.

Remarque importante : dans le format générique , il faut doubler les caractères ' pour qu'il soient interprétés comme de véritables simples quotes.

Exemples:

```
Object tabVal[] = new Object[3];

tabVal[0]=new Integer(2);
tabVal[1]= "oeufs";   tabVal[2]="poule";
String chRes = java.text.MessageFormat.format("La {2} a pondu {0} {1}.\n", tabVal);

tabVal[0]="Dupond";
tabVal[1]= new Integer(2);
String chRes2 = java.text.MessageFormat.format(
                "Select * from tableX where nom='{0}' AND num={1}",  tabVal);
```



5.3. classe abstraite NumberFormat et sous classe DecimalFormat

Exemple : Mise en forme d'un nombre réel avec seulement deux chiffres après la virgule:

```
NumberFormat nfmt = NumberFormat.getInstance();
nfmt.setMaximumFractionDigits(2);
nfmt.setMinimumFractionDigits(2);
double x=246.78934;
System.out.println("La valeur de x est : " + nfmt.format(x));
```

6. Nouveautés apportées par le jdk 1.8

Beaucoup de nouveautés du jdk 1.8 ont été influencées par les bibliothèques "guava" (de google) et "jodaTime" .

6.1. Interfaces fonctionnelles , "lambda expression" et références

Une **interface fonctionnelle** (que l'on peut facultativement explicitement marquée avec la nouvelle annotation **@FunctionalInterface** du **jdk >= 1.8**) est **une interface qui ne comporte qu'une seule méthode ordinaire** (sans "static" ni "default").

Son rôle est de permettre une gestion simple d'une certaine méthode/fonction abstraite .

(*SAM*= **S**ingle **A**bstract **M**ethod → interface de type "SAM")

Exemple (V1) :

```
package tp.langage.v8.sam;
/* L'annotation @FunctionalInterface (du jdk >= 1.8) est facultative
  elle permet  au compilateur de vérifier que l'interface comporte une seule méthode (ordinaire) */
@FunctionalInterface
public interface SamPredicate<E> {
   boolean test(E e); //retourne true si l'entité e (de type E) satisfait certains critères
}
```

Exemple (V2) :

```
package tp.langage.v8.sam;

@FunctionalInterface
public interface SamPredicate<E> {
   boolean test(E e); //retourne true si l'entité e (de type E) satisfait certains critères
   default String getAuthor() { return "developpeur fou"; } //méthode par défaut
     // (alias "extension method" alias "defender method" ). Les méthodes par défaut ont permis
     // d'ajouter de nouvelles fonctionnalités à java8 sans remettre en question les interfaces java7
   static void methodeStatiqueAutoriseeSurInterfaceDepuisJava8(String msg){
         System.out.println(msg);
       }
   //les méthodes statiques au sein des interfaces permettent d'éviter la programmation de
   // nombreuses mini classes utilitaires périphériques
}
```

NB : **java.util.function.Predicate** (depuis java 8) correspond à un **équivalent prédéfini** de la version 1 de l'interface fonctionnelle SamPredicate .

Classe "Person" (pour la compréhension de l'exemple) :

```
...
public class Person {
   String firstname; //+get/set
   String lastname; //+get/set
   int age; //+get/set

   public Person(String firstname, String lastname, int age) { ...}
   public Person(){ }
   public String toString() {return "Person ("+  firstname + "," + lastname + ", age=" + age + ")"; }

   public static int sortByAge(Person p1, Person p2) {
      if (p1.getAge() > p2.getAge()) {
         return 1;
      } else if (p1.getAge() < p2.getAge()) {
         return -1;
      } else {
         return 0;
      }
   }
}
```

Syntaxe lourde (java 7) avec classes anonymes imbriquées :

```
...
public class FilterSortListJava7TestApp {
          …
public static List<Person> extractSubListBySamPredicate(List<Person> pList,
                                                        final SamPredicate<Person> samPredicate)
{
      final List<Person> sublist = new ArrayList<Person>();
      for (Person p : pList) {
         if (samPredicate.test(p)) {
                  sublist.add(p);
         }
      }
      return sublist;
```

```
    }
    …

        public static void mainWithInnerAnonymousSamPredicateImplementations() {
                List<Person> pList = StaticPersonList.initList();

                System.out.println("person sublist with age from 20 to 25:");
                List<Person> subList1 = extractSubListBySamPredicate(pList,
                        new SamPredicate<Person>(){
                          @Override
                          public boolean test(Person p) {
                                  return p.getAge() >= 20 && p.getAge() <= 25;
                          }
                });
                System.out.println(subList1);
                System.out.println("person sublist with lastName starting with letter p");
                List<Person> subList2 = extractSubListBySamPredicate(pList,
                         new SamPredicate<Person>(){
                          @Override
                          public boolean test(Person p) {
                                  return p.getLastName().startsWith("p");
                          }
                });
                System.out.println(subList2);
        }
        …
}
```

Syntaxe (très allégée) avec lambda expression depuis de jdk 1.8 :

```
package tp.langage.v8.lambda;
….
import java.util.function.Predicate;

public class FilterSortListJava8TestApp {
...
```

```
public static List<Person> extractSubListByJava8Predicate(List<Person> pList,final
Predicate<Person> predicate) {
      final List<Person> sublist = new ArrayList<Person>();
      for (Person p : pList) {
        if (predicate.test(p)) {
              sublist.add(p);
        }
      }
      return sublist;
   }
public static void mainWithLambdaExpressions() {
        List<Person> pList = StaticPersonList.initList();
        System.out.println("person sublist with age from 20 to 25:");
        List<Person> subList1 = extractSubListByJava8Predicate(pList,
              (person) -> person.getAge() >= 14 && person.getAge() <= 25      );
        System.out.println(subList1);
        System.out.println("person sublist with lastName starting with letter p");
        List<Person> subList2 = extractSubListByJava8Predicate(pList,
              (person) -> person.getLastName().startsWith("p")    );
        System.out.println(subList2);
        }
…
}
```

La nouvelle syntaxe ***(arguments) -> corps de la fonction***
apportée par le jdk 1.8 est appelée "**lambda expression**" .

Derrière cette syntaxe très concise se cache un gros travail de déduction / résolution de la part du compilateur :

- fabrication automatique d'une classe anonyme respectant l'interface uni-fonctionnelle précisée.
- mise en rapprochement des arguments de la "lambda expression" avec les paramètres d'entrées de la méthode abstraite (de l'interface fonctionnelle)
- utilisation du corps de la fonction précisé au sein de la "lambda expression"
- …

→ les types des arguments peuvent ainsi être déduits (et leurs compatibilités vérifiées) par le compilateur .

Autre exemple :

```
public static void sortListComparatorInnerAnonymousImplementation() {
        List<Person> pList = StaticPersonList.initList();
        Collections.sort(pList, new java.util.Comparator<Person>(){
                @Override
                public int compare(Person p1, Person p2) {
                        if (p1.getAge() > p2.getAge()) {     return 1;   }
                         else if (p1.getAge() < p2.getAge()) {  return -1; }
                           else { return 0; }
                        }
                });
                System.out.println(pList);
        }
```

simplifié en

```
public static void sortListWithLambdaExpression() {
    List<Person> pList = StaticPersonList.initList();
   Collections.sort(pList,
   (Person p1, Person p2) -> p1.getAge() == p2.getAge() ? 0 : (p1.getAge() < p2.getAge() ? -1 : 1 ) );
  System.out.println(pList);
}
```

ou bien via

```
public static void sortListWithFunctionReference() {
        List<Person> pList = StaticPersonList.initList();
        Collections.sort(pList, Person::sortByAge);
              //Person::sortByAge() have same code as java.util.Comparator.compare()
              //but with different name and without explicit interface implementation
        System.out.println(pList);
        System.out.println("liste triée par nom puis par prénom (with function reference):");
        pList.sort(Comparator.comparing(Person::getLastName)
                                        .thenComparing(Person::getFirstName));
        System.out.println(pList);
        }
```

NB : la nouvelle syntaxe **NomClasse::nomFonction** correspond à une **référence de fonction** .
→ les arguments et le corps de la fonction référencée sont alors utilisés de la même façon que ceux d'une "lambda expression" .

NB2 : une référence de fonction peut éventuellement **référencer un "constructeur"** tel que le montre cet exemple :

```
....map(person -> new Student(person));
```

pouvant être ré-écrit en

```
...map(Student::new);
```

Scope visibility in lambda expression :

```
public class Java8TestApp {
        private String message="Hello World (V8)";
        public static void main(String[] args) {
                (new Java8TestApp()).testRun();
        }
        public void testRun(){
                //lambda expression (compatible run()) utilisant "message" du scope parent:
                Runnable r1 = () -> System.out.println(message);
                Thread t1 = new Thread(r1); t1.start();
        }
}
```

Avec un peu plus de recul et plus formellement :

Une **interface fonctionnelle** correspond à un **type (de traitement fonctionnel) abstrait**.

Une "**lambda expression**" (*ou bien une référence de méthode*) correspond à **une implémentation concrète d'une interface fonctionnelle** .

→ on peut écrire

Runnable r1 = () -> System.out.println("message");

D'autre part, le package **java.util.function** comporte un paquet d'**interfaces uni-fonctionnelles génériques** :

Function<T,R> - takes an object of type T and returns R.

Supplier<T> - just returns an object of type T.

Predicate<T> - returns a boolean value based on input of type T.

Consumer<T> - performs an action with given object of type T (no return)

BiFunction - like Function but with two parameters.

BiConsumer - like Consumer but with two parameters

Exemple d'utilisation (explicite et directe) :

```
...
List<String> liste = initListOfString();
Function<String, String> atrFct = (name) -> {return "@" + name;} ;
Function<String, Integer> lengthFct = (name) -> name.length()  ;
//ou bien Function<String, Integer> lengthFct = String::length ;
for (String s : liste) {
   System.out.println( atrFct.apply(s) + " , length=" + lengthFct.apply(s));
}
```

L'utilisation des types fonctionnels abstraits et génériques (de **java.util.function**) est souvent effectuée indirectement en tant que paramètres des opérations (sort , filter , map , …) sur les "Streams" .

6.2. Optional<T>

java.util.Optional (depuis le jdk 1.8) correspond à un **conteneur (jamais "null") de valeur "objet" optionnelle** . Ceci un sémantiquement plus précis qu'une valeur "null" et est en autres utilisé comme type de retour de certaines versions de stream.**reduce**(...) .

Exemple :

```
...
public static List<String> initWinnerList(){
        String[] winnerArray = {"bob", "anna", "alice"};
        List<String> winnerList = Arrays.asList(winnerArray);
        return winnerList;
}
public static Optional<String> getWinnerByName(String name){
        for(String s : initWinnerList()){
                if(s.equals(name))
                        return Optional.of(s);
        }
        /*else*/
        return Optional.empty(); //instead of return null
}
public static void testOptional(){
        Optional<String>  opStr = getWinnerByName("alice");
        System.out.println("Optional<String> opStr = " + opStr );
        if(opStr.isPresent())
           System.out.println(opStr.get());
```

```
        //with lambda expression:
        opStr.ifPresent( (s) -> System.out.println(s) );

        opStr = getWinnerByName("looser");
        System.out.println("Optional<String> opStr = " + opStr );
        //System.out.println(opStr.get());
        //java.util.NoSuchElementException instead of nullPointerException
        opStr.ifPresent((s)->System.out.println(s));

        //String str = "abc";
        String str = null;
        Optional<String> opS = Optional.ofNullable(str); //build .empty() if null
        System.out.println("Optional<String> opS = " + opS );

        String msg = opS.map((notNullStr)-> "not null string value : " + notNullStr)
                        .orElse("empty optional");
        System.out.println(msg);
    }
```

6.3. Stream et opérations (sort , filter , map , reduce , collect , ...)

java.util.stream.Stream (depuis le jdk 1.8) permet d'effectuer **des opérations de tri , filtrage , ... avec une syntaxique très concises sur des flux de données en vrac** *(bulk data operations in english).*

Exemple:

```
public static void mapWithStream(){ // map() is for "transformation"
        List<Person> persons = StaticPersonList.initList();
        Stream<Student> studentsStream = null;
        /*
        //v1 (explicit):
        studentsStream = persons.stream().filter(p -> p.getAge() <= 30)
                .map(new Function<Person, Student>() {
                    @Override
                    public Student apply(Person person) {
                      return new Student(person);
                    }
                });
```

```
        //v2 (with lambda expression):
        studentsStream = persons.stream().filter(p -> p.getAge() <=30 )
              .map(person -> new Student(person));
        */
        //v3 (with function/constructor reference):
        studentsStream = persons.stream().filter(p -> p.getAge() <=30 )
              .map(Student::new);
        System.out.println("liste of  students:");
        studentsStream.forEach(System.out::println);



        //with collect() terminal operation (at end of operation stream) :
        List<Student> students =  persons.stream().filter(p -> p.getAge() <=25 )
              .map(Student::new)
              .collect(Collectors.toList()); //or .collect(Collectors.toCollection(ArrayList::new));
        System.out.println("liste of (youngs) students:" + students);
        }
```

```
public static void skipAndLimitOnStream(){
    final List<Integer> demoValues  = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
    //limit the input -> [1, 2, 3, 4] :
    System.out.println(demoValues.stream().limit(4).collect(Collectors.toList()));
    //jumping over the first 4 elements -> [5, 6, 7, 8, 9, 10] :
    System.out.println(demoValues.stream().skip(4).collect(Collectors.toList()));
      }
```

```
public static void reduceStreamToASingleResult(){
    final List<String> demoValues  = Arrays.asList("1_2_3","-4_5_6","-7_8_9");
    System.out.println("stream reduced to as single optional result: "
                    + demoValues.stream().reduce(String::concat));

    final List<Integer> demoValues2  = Arrays.asList(5,8,12);
    System.out.println("stream reduced to as single result (somme des valeurs): "
            + demoValues2.stream().reduce(0 , (x,y) -> x+y));
            //.reduce(identity, accumulator) where identity is initialValue
            // (or default result if stream is empty)
```

```
    }
```

```
public static void filteringWithStreamFilterAndLambda(){
    List<Person> pList = StaticPersonList.initList();
    pList.stream().filter( (p) -> p.getAge() >= 32  ).forEach( (p) -> System.out.println(p) );

    List<String> liste = initListOfString();
    //liste.stream().map((s)->s.toUpperCase()).forEach(System.out::println);
    liste.stream().map(String::toUpperCase).forEach(System.out::println);

    //.stream() for sequential operations ,  .parallelStream() for parallel operations
    //source of stream can be:
    //Stream.of(val1,val2,val3…) , Stream.of(array) and list.stream().
    //or final Stream<String> splitOf = Stream.of("A,B,C".split(","));
    }
```

Autres opérations possibles(sur les streams) :

.**distinct**()

.**sequential**() , .**parallel**()

.**sorted**(*comparator_lambda_expression*)

.**findFisrt**() , ...

.**max**() , .**min**() , .**average**()

.**mapToInt**(Integer::parseInt)

.**mapToObj**("a" + i)

.**peek**() , .**flatMap**() , .**anyMatch**() , **noneMatch**() , ….

...

à approfondir avec internet et/ou la documentation officielle (javadoc) .

6.4. LocalTime , LocalDate (de java.time)

*java.time.***LocalTime** , **LocalDate** (d'inspiration JodaTime) sont plus simples à utiliser que **Date** et **Calendar**

Exemple :

```
public static void testLocalTimeAndLocalDate(){
LocalTime now = LocalTime.now();        System.out.println("now is " + now);
LocalTime later = now.plus(8, ChronoUnit.HOURS);
System.out.println("later (now+8h) is " + later);


LocalDate today = LocalDate.now();        System.out.println("today is " + today);
LocalDate thirtyDaysFromNow = today.plusDays(30);
System.out.println("thirtyDaysFromNow is " + thirtyDaysFromNow);
LocalDate nextMonth = today.plusMonths(1); System.out.println("nextMonth is " + nextMonth);
LocalDate aMonthAgo = today.minusMonths(1); System.out.println("aMonthAgo is " + aMonthAgo);


//LocalDate date14July2015 = LocalDate.of(2015, 7, 14);
LocalDate date14July2015 = LocalDate.of(2015, Month.JULY, 14);


LocalTime time = LocalTime.of(14 /*h*/, 15 /*m*/ , 0 /*s*/);
LocalDateTime datetime = date14July2015.atTime(time);
System.out.println("le 14 juillet 2015 à 14h15 : " + datetime);


LocalDate date1=today , date2=nextMonth;
Period p1 = Period.between(date1, date2) ;    System.out.println("periode p1:" + p1);


LocalTime time1= time;
LocalTime time2= LocalTime.of(14 /*h*/, 30 /*m*/ , 0 /*s*/);
Duration d = Duration.between(time1, time2);      System.out.println("durée d:" + d);


Duration twoHours = Duration.ofHours(2); System.out.println("durée de 2 heures:" + twoHours);
Duration tenMinutes = Duration.ofMinutes(10); System.out.println("durée de 10 minutes:" + tenMinutes);
Duration thirtySecs = Duration.ofSeconds(30); System.out.println("durée de 30 secondes:" + thirtySecs
LocalTime t2 = time.plus(twoHours);        System.out.println("14h15 plus 2 heures:" + t2);


//.with(temporalAdjuster) :
LocalDate premierJourDeCetteAnnee=
LocalDate.now().with(TemporalAdjusters.firstDayOfYear());
```

```
System.out.println("premierJourDeCetteAnnee="+premierJourDeCetteAnnee);
LocalDate dernierJourDuMois=  LocalDate.now().with(TemporalAdjusters.lastDayOfMonth());
System.out.println("dernierJourDuMois="+dernierJourDuMois);

ZoneId myZone = ZoneId.systemDefault();
System.out.println("my (local) zoneId is:" + myZone);

//lien entre java.util.Date et java.time... :
Date date = new Date();
Instant nowInstant = date.toInstant();
LocalDateTime dateTime = LocalDateTime.ofInstant(nowInstant, myZone);
System.out.println("today/now from Date:" + dateTime);
}
```

6.5. Encodage base64 (java.util.Base64)

```
public static void testBase64(){
        try {
        // Encode using basic encoder :
        String base64encodedString =
                Base64.getEncoder().encodeToString("Myjava8String".getBytes("utf-8"));
        System.out.println("Base64 Encoded String (Basic) :" + base64encodedString);

        // Decode :
        byte[] base64decodedBytes = Base64.getDecoder().decode(base64encodedString);
        System.out.println("Original String: " + new String(base64decodedBytes, "utf-8"));
        } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
        }
}
```

6.6. Diverses autres nouveautés de java 8

Amélioration de l'introspection si option "**-arguments**" au lancement du compilateur "**javac**"

→ parameter.getName() retourne véritable nom du paramètre (stocké dans byteCode) plutôt que "arg0" , "arg1" , …

@Repeatable (java.lang.annotation) , ...

VI - Entrées/sorties (io) - fichiers

1. Lecture / écriture dans un fichier et à l'écran.

Les classes abstraites **InputStream** et **OutputStream** définissent les méthodes de bases permettant d'effectuer des entrées/sorties : **.read()** , **.write(), .flush(), .close(),** …

Ces classes génériques sont spécialisées en fonction du support :

FileInputStream / **FileOuputStream** pour lire ou écrire dans un fichier
ByteArrayInputStream / ByteArrayOutputStream ---> tableau d'octets en mémoire
PipedInputStream / PipedOutputStream --- > tuyau de communication entre 2 threads

Avec les classes ci-dessus, on peut effectuer des entrées/sorties de bas niveau (flux d'octets brut sans notion de format).
Pour contrôler plus finement le format des données à lire ou à écrire, on sera généralement amené à instancier à partir de ces flux bruts, l'une des classes suivantes:

DataOutputStream	Pour écrire **en binaire** des entiers, réels, …
PrintStream	Pour écrire sous forme de **chaînes ASCII** des String,int,…
BufferedReader & InputStreamReader	pour lire des chaînes de caractères que l'on récupérera sous forme de String,int,…

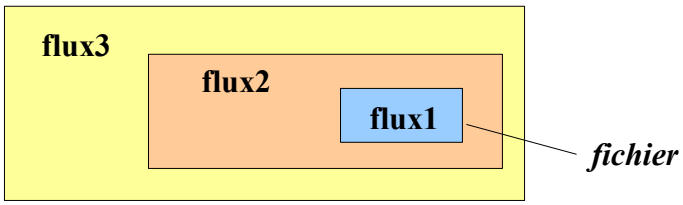
Les entrées/sorties standards (descripteurs Unix 0,1,2) sont depuis JAVA manipulées au moyen de **System.in** , **System.out** et **System.err** qui sont des instances statiques (variables de classe) intégrées au sein de la classe System .

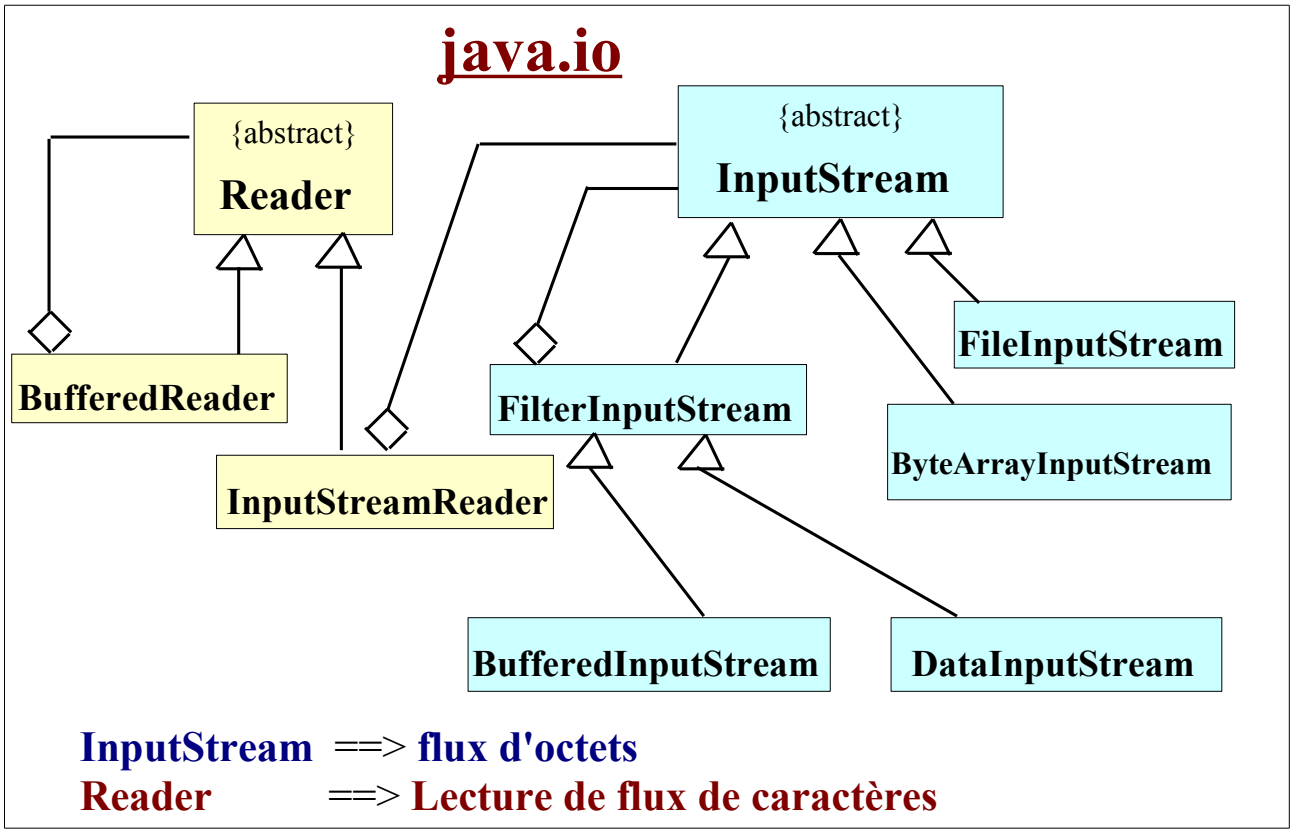
2. Principe fondamental d'imbrication des flux

Le package **java.io** est organisé en suivant le principe du **modèle de conception** (Design Pattern) "**Décorateur**" qui associe *héritage* et *aggrégation* .

Quelque soit le type de flux d'octets manipulé, celui-ci peut être vu comme étant une chose de type ***OutputStream*** ou bien ***InputStream***.

Pour obtenir des fonctionnalités spécifiques , il faut imbriquer un flux de bas niveau dans un flux de haut niveau:





Exemple: pour lire des lignes dans un fichier texte , il faut commencer par récupérer un flux de bas niveau en lecture sur le fichier:

```
FileInputStream flux1 = new FileInputStream("c:\\repx\\f1.txt");
```

Cet flux ne comporte qu'une simple méthode ***read***() permettant de ***lire un paquet d'octets en binaire.***
On imbrique donc ce flux dans un autre de plus haut niveau qui va apporter des fonctionnalités supplémentaires (Reconnaissance des caractères et ***lecture caractère par caractère***):

```
InputStreamReader flux2 = new InputStreamReader(flux1);
```

Ceci ne nous permet toujours pas de lire facilement des lignes entières.
On imbrique donc ce flux intermédiaire dans un troisième flux d'encore plus haut niveau et qui sera capable de "bufferiser" les caractères lus , reconnaître des fins de lignes et retourner des ***lignes entières*** via la méthode supplémentaire ***readLine*** () :

```
BufferedReader flux3 =  new BufferedReader(flux2);
```

3. Exemple de code

Cet exemple permet de saisir une phrase à partir de l'entrée standard. Cette ligne est alors introduite dans un fichier f1.txt.
Ce fichier est ensuite intégralement relu puis ré affiché à l'écran.

```
import java.io.*;

class AppliModeTexte {

public static void main(java.lang.String[] args) {
try {
      System.out.println("Entrez une valeur :");
      BufferedReader dts = new BufferedReader(
               new InputStreamReader(System.in /*flux d'entrée brut*/));

      String s = dts.readLine();
      System.out.println("-----> Valeur saisie :" + s);


      // Ecriture de quelques lignes dans le fichier "f1.txt"

      FileOutputStream of = new FileOutputStream("f1.txt");
      PrintStream ps = new PrintStream(of);

      ps.println("Ligne 1 (debut)");
      ps.println("Valeur saisie : " + s );
      ps.println("Ligne 3 (fin)");
      // fermetures dans l'ordre inverse des ouvertures:
      ps.close(); of.close();

      //Relecture de ce fichier et affichage des lignes
      //sur la sortie standard

      FileInputStream ifile = new FileInputStream("f1.txt");
      BufferedReader dis =
            new BufferedReader(new InputStreamReader(ifile));

      while (true)
         {
         s = dis.readLine();  // lecture d'une ligne dans le fichier
         if(s!=null) // Réécriture de celle si sur la sortie standard
             System.out.println(s);
            else break;
         }
      dis.close(); ifile.close();

      } /* end of try */

catch(java.io.IOException e)
      {
      System.err.println("Exception I/O"); e.printStackTrace();
      }
   }/* end of main */
}
```

4. (File) répertoires et des attributs sur les fichiers

La classe java.io.**File** permet de gérer les répertoires et les fichiers d'une manière indépendante de la plate-forme (PC, Unix,…).

Après avoir instancier à partir d'un nom de fichier ou de répertoire la classe File on utilisera généralement l'une des principales méthodes suivantes:

getName()	Nom final sans nom de répertoire
getPath() , getAbsolutePath()	Chemin relatif ou absolu menant au fichier
getParent()	Répertoire contenant l'objet courant
isAbolute() , exists()	relatif/Absolu ? Nom correct ?
canRead() , canWrite()	Pour tester les permissions d'accès
isFile() , isDirectory()	De quoi s'agit-il ?
length()	Longueur du fichier en octets
String[] **list(), … list(filtre)**	Retourne une liste de fichiers
renameTo(String newName**)**	Renomme la chose
mkdir()	Créer un nouveau répertoire
delete()	Efface la chose

L'exemple ci-dessous permet de récupérer la liste des fichiers d'extension "xml" situés dans un certain répertoire:

```
File f = new File(dirName);
XmlFileFilter filter = new XmlFileFilter();
String tabFile[] = f.list(filter);
int nbf=tabFile.length;
...
```

avec:

```
class  XmlFileFilter implements FilenameFilter
{
public boolean accept(File dir,String name)
  {
   int n = name.length();
   if(n <5) return false;
   String ext=name.substring(n-3,n);
   if(ext.toLowerCase().equals("xml")) return true;
     else return false;
  }
}
```

NB: bien que les fonctionnalités de cette classe soient fort intéressantes , il ne faut pas hésiter à invoquer une instance de la classe prédéfinie java.awt.**FileDialog** (ou bien la nouvelle version javax.swing.**FileChooser**) de façon à choisir un nom de fichier à ouvrir ou à sauvegarder.

5. Spécificités à connaître et détails intéressants

NB: pour des raisons de sécurité, un applet (non signé) ne peut pas manipuler les fichiers de la machine sur laquelle il a été téléchargé .
Par contre une application autonome peut à loisir effectuer toutes les opérations classiques (Création, lecture, écriture,…) sur les fichiers et les répertoires accessibles.

Les classes (filtres) **GZIPOutputStream** et **GZIPInputStream** du package **java.util.zip** permettent de gérer le **format de compression gzip** .

exemple:
...
GZIPOutputStream outputGZipStream = new GZIPOutputStream(fileStream);
outputGZipStream.write(....);
outputGZipStream.close();

6. Fichier de données accompagnant le code

Lorqu'une application java a besoin de charger en mémoire un fichier de données qui est déployé avec son code (au même endroit que le code compilé : répertoire ou fichier ".jar"), il est conseillé de procéder de la façon suivante:

InputStream fluxLecture = **getClass().getClassLoader().getResourceAsStream**("*ficData.txt*");

car le chemin relatif "*ficData.txt*" est exprimé par rapport à la racine du code et ne devrait pas changer (même si tout est recopié sur une autre machine).

7. Simplification des flux d'entrées - Java 5

Exemple de code java (toute version) permettant la lecture d'un entier au clavier :

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String str = br.readLine();
int n = Integer.parseInt(str);
```

La nouvelle classe **java.util.Scanner** permet le même traitement avec moins de code :

```
Scanner reader = new Scanner(System.in);
int n = reader.nextInt();
// String ch = reader.next();
```

Pour gérer des entrée plus complexe, on peut utiliser la classe ***java.util.Formatter***, qui inclus des algorithmes à base de patterns.

VII - Introspection et Sérialisation

1. Introspection (java.lang.reflect)

Introspection

Le package **java.lang.reflect** permet d'effectuer une **introspection** des classes java. L'introspection consiste à ***demander aux classes de dresser la liste de leurs attributs et méthodes***.

Cette **faculté d'auto-analyse** qu'offre les mécanismes internes du langage Java est pour l'instant inexistante en C++ .
➔L'introspection est un **gros point fort de Java** .
➔Ceci ***permet d'automatiser entièrement des opérations de bas niveaux (sauvegarde / restauration de valeurs, ...).***

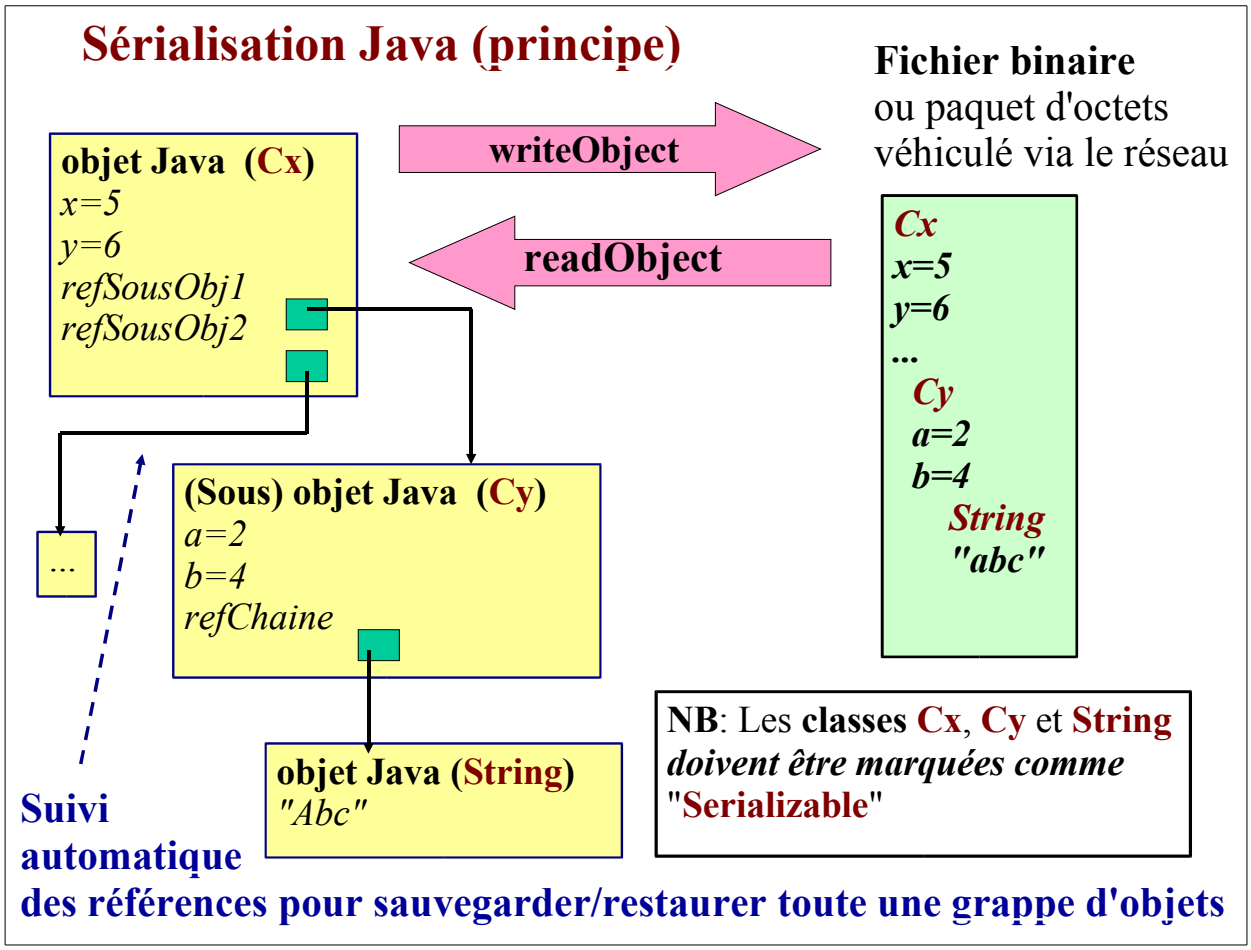
```
import java.lang.reflect.*;

Class c = Class.forName(nomClasse);
Field[] tabChamps = c.getDeclaredFields();
Method[] tabMethods = c.getDeclaredMethods();
 + boucle "for(i=0; i < tabXxxx.length; i++) ...
```

Une classe java se prête alors assez bien à certaines sortes de ***programmation générique***:

- Un IDE (Environnement de Développement Intégré) peut, à partir de valeurs saisies dans une fenêtre de propriétés, générer les lignes de code source du type NomObjetX.setNomAttribut(ValeurChoisie) .

- Un langage de script (tel que javascript) peut interroger une classe Java (ex: un applet se conformant aux spécificités "JavaBean") pour :
  - référencer un attribut d'un nom donné de façon à changer sa valeur (via setXxx() de la classe **Field**).
  - référencer une méthode d'un nom donné de façon à l'invoquer(via **invoke**() de la classe **Method**).

2. Sérialisation (et persistance élémentaire)



2.1. Problématique de la persistance :

Un objet (instance) Java vit en mémoire et sa durée de vie est éphémère (le temps de l'exécution de l'application au maximum).

En programmation orienté objet, la **persistance** désigne le fait de pouvoir **stocker les valeurs d'un objet entier dans un flux** (fichier local, base de données, ...) **de façon à pouvoir , plus tard , restaurer les valeurs de cet objet dans une nouvelle instance en tous points identique à l'instance d'origine**.

D'un point de vue technique, la persistance est un peu plus compliquée qu'un simple dump binaire puisqu'**il faut suivre les références et sauvegarder tous les objets liés à l'instance courante (objet principal)**.

D'autre part, lorsque l'on doit recréer une instance à partir des données que l'on extrait du flux, **il faut tenir compte des types (classes) exacts des objets liés qu'il faut reconstruire, restaurer et rattacher** à l'objet principal.

Lorsque l'on utilise les mécanismes prédéfinis du langage ou d'une API pour gérer la persistance, on a généralement affaire à un **code très simple** mais en contre partie, **on ne contrôle pas vraiment le format de ce qui est sauvegardé** et l'on doit faire attention aux **problèmes liés à des décalages d'octets dus à des versions différentes de la classe**.

2.2. Mécanisme de persistance élémentaire du langage JAVA

Les classes **ObjectOutputStream** et **ObjectInputStream** comportent respectivement les méthodes **readObject** et **writeObject** permettant de gérer automatiquement la persistance des objets java.

NB: pour pouvoir appeler **readObject()** ou **writeObject()** sur un objet JAVA, il faut que la classe (dont est issue l'instance) implémente l'interface java.io.**Serializable** .

Le mécanisme de persistance ainsi déclenché fera en sorte que :

- Les attributs marqués via le mot clef "**transient**" ne soient pas sauvegardés, ni restaurés.
- Les attributs "**static**" (liés à la classe et non pas aux instances) ne soient pas traités.

Sérialisation (exemple de code)

```
import java.io.*;
----------------------
FileOutputStream ofStream = new FileOutputStream(pathName);
ObjectOutputStream ofluxObj = new  ObjectOutputStream(ofStream);
ofluxObj.writeObject(monObjet);
----------------------
FileInputStream ifileStream = new  FileInputStream(pathName);
ObjectInputStream ifluxObj = new  ObjectInputStream(ifileStream);
monObjet = (MaClasse) ifluxObj.readObject();
----------------------
public class MaClasse implements java.io.Serializable
{
private int x,y;
private String nom;
...
}
```

Remarque:

- L'interface **Serializable** ne comporte aucune fonction imposée.
  C'est une interface de marquage (uniquement utilisée pour tester le type).

- La notion de sérialisation n'est pas limitée au support "fichier".
  On peut stocker un ensemble d'objets Java dans un flux quelconque.

- L'api **RMI** (Remote Method Invocation) **utilise de façon transparente** la **sérialisation** pour **transférer** un ensemble d'objets du serveur vers le client ou vice-versa .

2.3. Gestion des versions

Les mécanismes internes du langage Java génère automatiquement une valeur par défaut (selon un algorithme lié à la compilation) pour le champ spécial **serialVersionUID (ajouté automatiquement à la classe s'il n'est pas présent au sein du code source) .** Ceci permet de lever automatiquement une exception de type `InvalidClassException` en cas d'incohérence sur le numéro de version .

NB: Il est grandement conseillé d'introduire explicitement la variable de classe **serialVersionUID** (de type **long**) au sein d'une classe sérialisable pour bien contrôler le numéro de version des objets qui seront lus ou écrits:

```
class Cx   implements   java.io.Serializable
{
 static final long serialVersionUID = 1L;
....
 }
```

2.4. Autres mécanismes de persistance

L'interface **java.io.Serializable** (et les méthodes ***readObject***() / ***writeObject***() de la classe **java.io.ObjectOutputStream**) correspondent au mécanisme de **persistance élémentaire** du langage JAVA.

Il existe d'autres mécanismes de persistance dans le monde Java:

- L'API "**JAXB**" (***Java Api for Xml Binding***) permet de gérer une persistance au format XML
- L'API "**Hibernate**" permet de gérer une persistance au sein d'une base de données relationnelle (MySQL , Oracle , ...)
- ...

VIII - Threads (java)

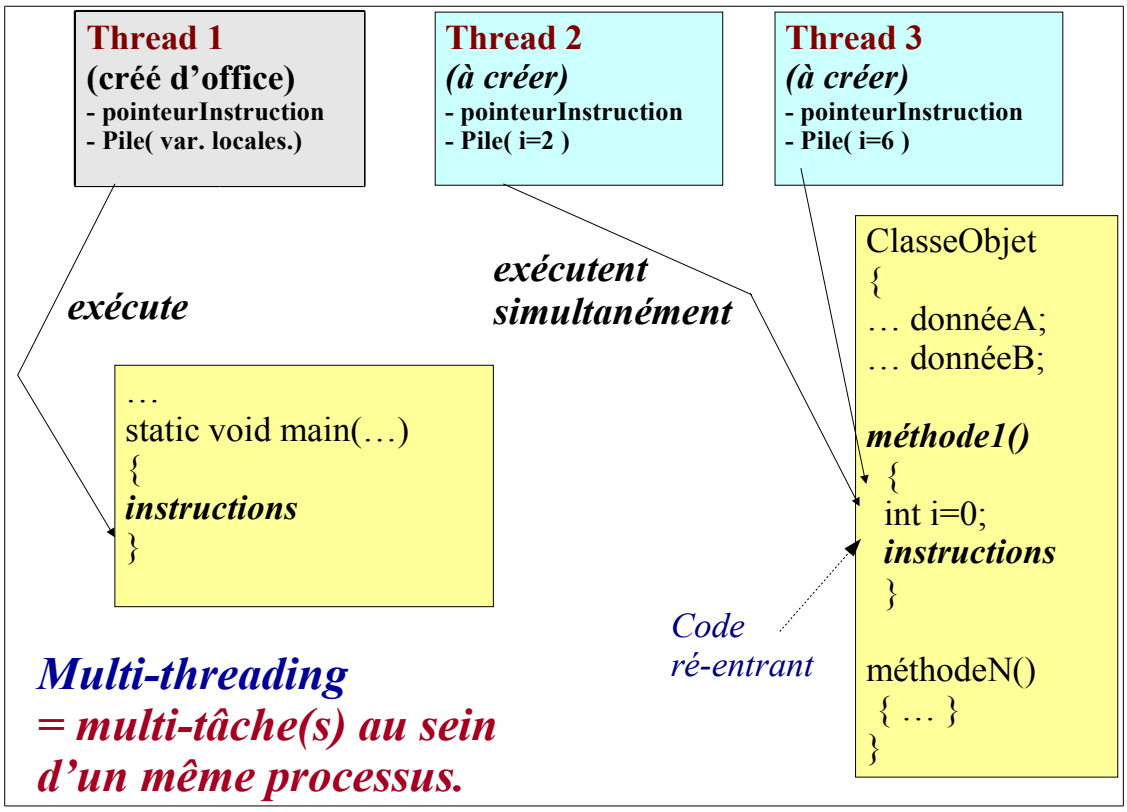
1. Concept de threads

Concept de Thread

Un **thread** est un **fil d'exécution** . Il s'agit d'une **entité dynamique qui peut exécuter un ensemble d'instructions**. L'association de *plusieurs threads* permet de mettre en œuvre **des traitements parallèles au sein d'un même processus** .

Sous Java, la gestion des threads fait intervenir trois entités fondamentales:

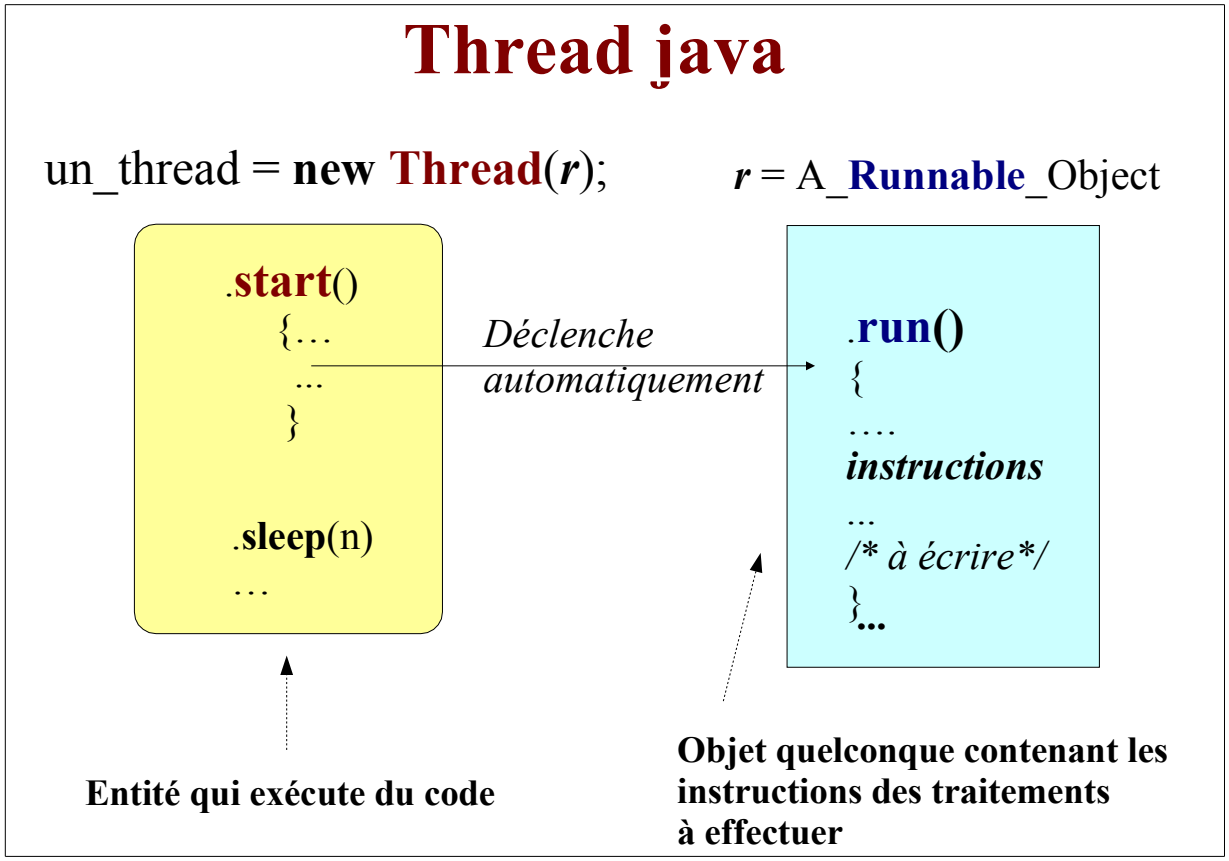
- Le ***moniteur de threads*** (gestionnaire central et caché dans les API de JAVA qui ***gère les exécutions concurrentes des différents threads***).
- ***Une instance de la classe Thread*** (***entité qui exécute du code*** et que l'on peut lancer, arrêter,…)
- ***Un objet implémentant l'interface Runnable*** et dont la méthode ***run*** correspond au point d'entrée de ***l'ensemble des instructions à exécuter*** par un nouveau thread.



Remarque: Il existe toujours au moins un thread au sein d'une application ou applet JAVA. Ce thread principal est créé automatiquement , il s'occupe de la gestion des événements associés à l'interface graphique ou bien de l'enchaînement des instructions partant de la méthode statique main() en mode texte. Si on fait exécuter par ce thread principal et implicite des traitements longs (boucle infinie, accès réseau, calculs , …) , on risque alors de bloquer la gestion de l'interface graphique et l'utilisateur aura alors l'impression que le programme ne répond plus.

Il est donc nécessaire de créer un ou plusieurs threads supplémentaires dès que l'on veut effectuer des opérations qui peuvent être longues ou bloquantes (Animations, Accès réseau, …).

2. Gestion des threads avec java



La méthode statique ***sleep***() est souvent employée dans les animations, elle permet d'insérer des temporisations entre deux affichages d'image par exemple.

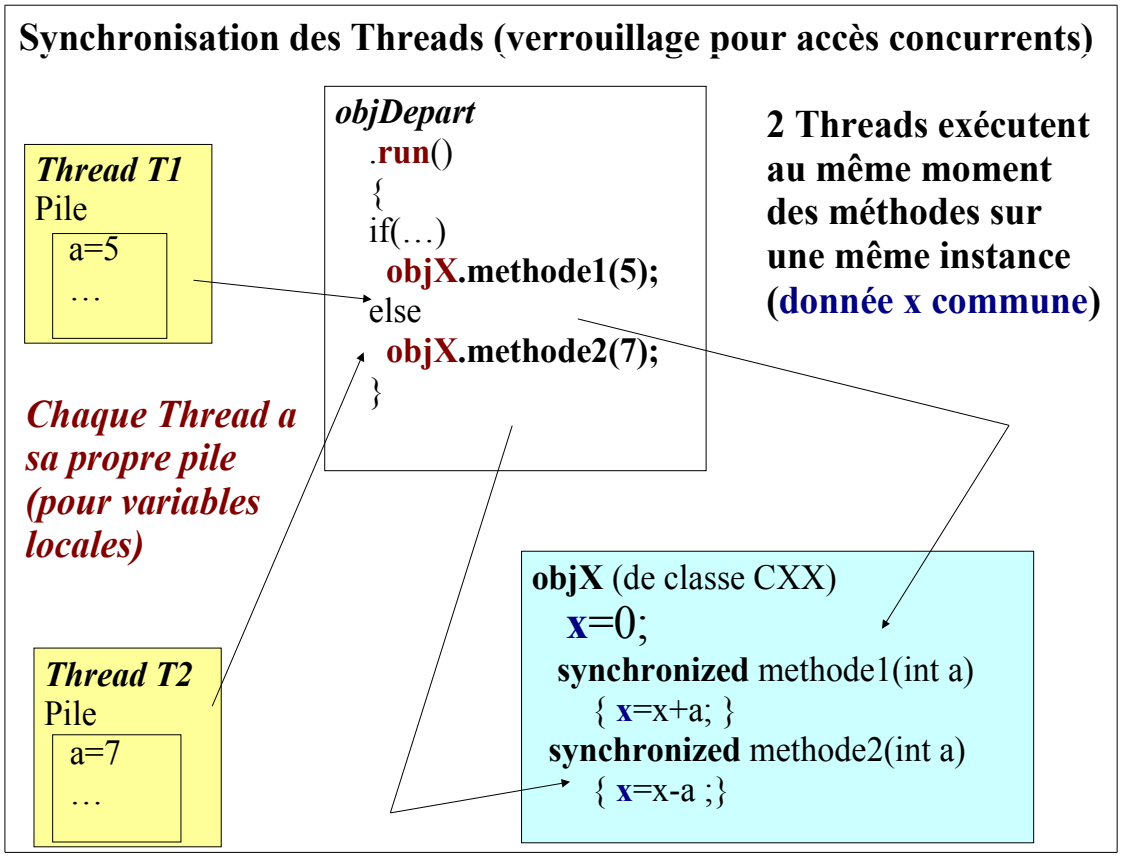
```
try{
        Thread.sleep(n); //méthode statique qui endort Thread.currentThread()
 } catch(InterruptedException ex) { ex.printStackTrace(); }
```

NB (Alternative) : Au lieu de créer une classe implémentant l'interface Runnable, **on** peut également **sous classer la classe Thread** et **programmer directement la méthode run() dans cette sous classe (Thread spécifique)**.

Exemple de code: (Applet avec animation)

```
public class MonApplet extends Applet implements Runnable
{
private boolean fin = false ; // variable commune vue de tous les threads.

public void init() { ... }

public void start()          // <--- méthode start() de la classe Applet (appelée automatiquement après init() )
{ demarrer_thread(); }

public void stop() /* <-- méthode stop() de la classe Applet (appelée juste
                      avant la méthode destroy() de l'applet quand on quitte la page WEB) */
{ arreter_thread() ; }


private void demarrer_thread()
{
Thread nouveauThread = null;  // thread secondaire qui va exécuter run()
 nouveauThread= new Thread(this);
 fin =false;
 nouveauThread.start(); //  <-- méthode start() de la classe Thread
}

private void arreter_thread()
{ fin=true; }


public void run()  // <-- méthode run() de l'interface Runnable et implémenté par l'applet lui même.
{
...
while (!fin)
        {
         try
            {  ...
           Thread.sleep(150);
           ...  }
        catch (InterruptedException e)
           { e.printStackTrace();  }
       }
// fin de la méthode run()
// ==> plus aucune instruction à exécuter ==> le thread s'arrête de lui même
// ==> la mémoire occupée par le thread terminé sera libérée quand il ne sera plus référencé.
}

}// fin de la classe de l'applet
```

3. Synchronisation des threads



Accès concurrents / synchronisation des threads

Dans certains cas , il se peut que ***plusieurs threads*** puissent ***exécuter simultanément différentes méthodes sur une même instance et donc manipuler les mêmes données internes de l'objet***.

Pour que toutes ces opérations puissent s'effectuer sans interférence, il faut temporairement verrouiller ces données pour qu'un seul thread puisse les modifier à la fois.

Le langage java a choisi de gérer cette synchronisation via un mot clef **synchronized** qui ***permet au moniteur central de la machine virtuelle de gérer la concurrence entre les threads*** :

Dès qu'un thread exécute une méthode synchronisée sur un objet java, tous les autres threads qui souhaitent exécuter une (même ou autre) méthode synchronisée sur la même instance seront automatiquement bloqués (mis en attente).

Le mot clef **synchronized** place donc une sorte de **verrou** sur un **objet complet** (pour protéger l'accès aux données internes).

Syntaxe:

Lorsque le mot clef ***synchronized*** est utilisé pour préfixer une méthode , l'objet (this) de la classe courante est alors automatiquement verrouillé durant tout le temps d'exécution de la méthode:

```
public class Cx {
protected int x=0;
synchronized public void methode1(int a)
        {
        x=x+a;
        ...
        }
 ...
}
```

Cas particulier des méthodes statiques:

Lorsque le mot clef **synchronized** est placé devant une méthode "**static**" , le verrou porte sur les membres statiques de la classe .

Lorque la syntaxe ***synchronized(Object) { ...}*** est utilisée à l'intérieur d'une méthode , seul l'objet (ou le sous objet) précisé est verrouillé et la durée du verrou correspond au temps nécessaire à un thread pour exécuter le bloc d'instructions délimité par les accolades :

```
public class Cy {
protected  CPartie  objPartX = new CPartie();
protected int y=0;

public void  methodeOrdinaire(...)
        {
        ...
        synchronized( objPartX )
                {
                objPartX.x++;
                ...
                }
        ...
        synchronized( this )
                {
                this.y++;
                ...
                }
        }
}
```

4. Attente et rendez-vous (wait & notify)

4.1. mécanismes

La classe **Object** qui est la racine de toutes les autres classes de Java comporte (entre autres) les fonctions **wait(timeout)** , **notify**() et **notifyAll**() .
NB: Les méthodes **wait**() et **notify**() ne peuvent êtres appelées sur une instance que si elles sont placées dans un bloc "**synchronized**" .

Comportement induit:

Un thread qui appelle la méthode **wait**(*timeout*) sur une instance sera mis en sommeil (pas de temps CPU consommé) jusqu'à ce que:

- La durée du **timeout** (en ms) ait expirée.

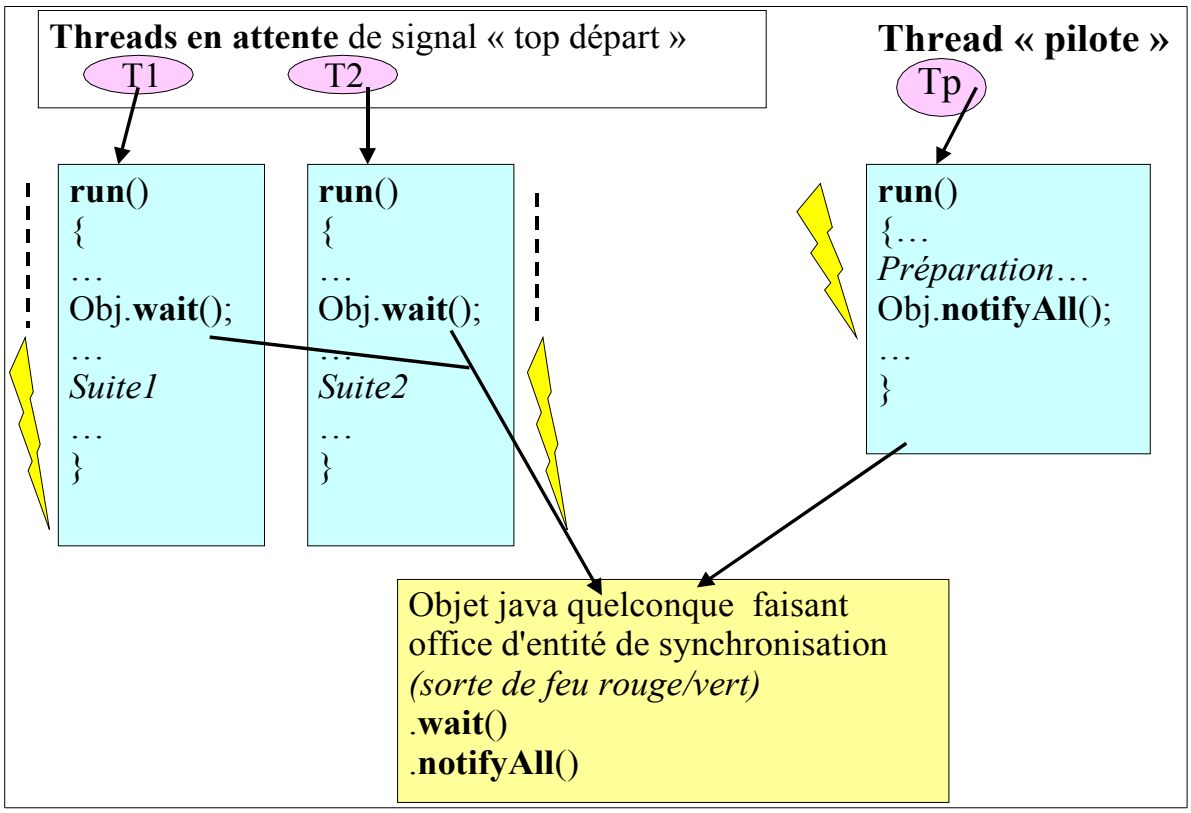
ou bien

- Un autre thread a appelé la méthode **notify**() sur le même objet.

ou bien

- Le thread en question a été interrompu (**InterruptedException**) .

NB: Lorsqu'un thread appelle la méthode **notify**() sur une instance, Le superviseur des threads ne réveille alors q'un seul autre thread (qui avait appelé wait()). La méthode **notifyAll**() permet de réveiller d'un coup tous les threads qui ont appelé la méthode wait() sur cette même instance.



4.2. Exemple concret ==> Gestion d'un pool de ressources

getRessource() avec un wait() sur un sous objet "Attente" s'il faut attendre une ressource disponible

libérerRessource() avec un notify() sur un sous objet "Attente" pour débloquer une seule des attentes

5. Autres aspects avancés sur les threads

- On peut donner un nom explicite (chaîne de caractères) à un Thread. Celui-ci peut être passé au niveau du constructeur.
- **join**() de la classe *Thread* permet d'attendre la fin de l'exécution d'un thread.
- Les Threads peuvent être organisés en groupe(s) et sous groupe(s).
- On peut modifier la **priorité** d'un thread : un_thread.**setPriority**(n);
- La classe **ThreadLocal** permet de gérer des variables qui sont accessibles globalement par une grande partie du code source et qui sont locales vis à vis des Threads (chaque Thread a sa propre valeur).

Le **jdk 1.5** a apporté un bon nombre de nouvelles fonctionnalités pointues sur les threads (ex: **sémaphore** du package ***java.util.concurrent***).

IX - JDBC (accès aux bases de données)

1. JDBC : Présentation et structure

Présentation de JDBC

• **JDBC** signifie ***Java DataBase Connectivity*** .

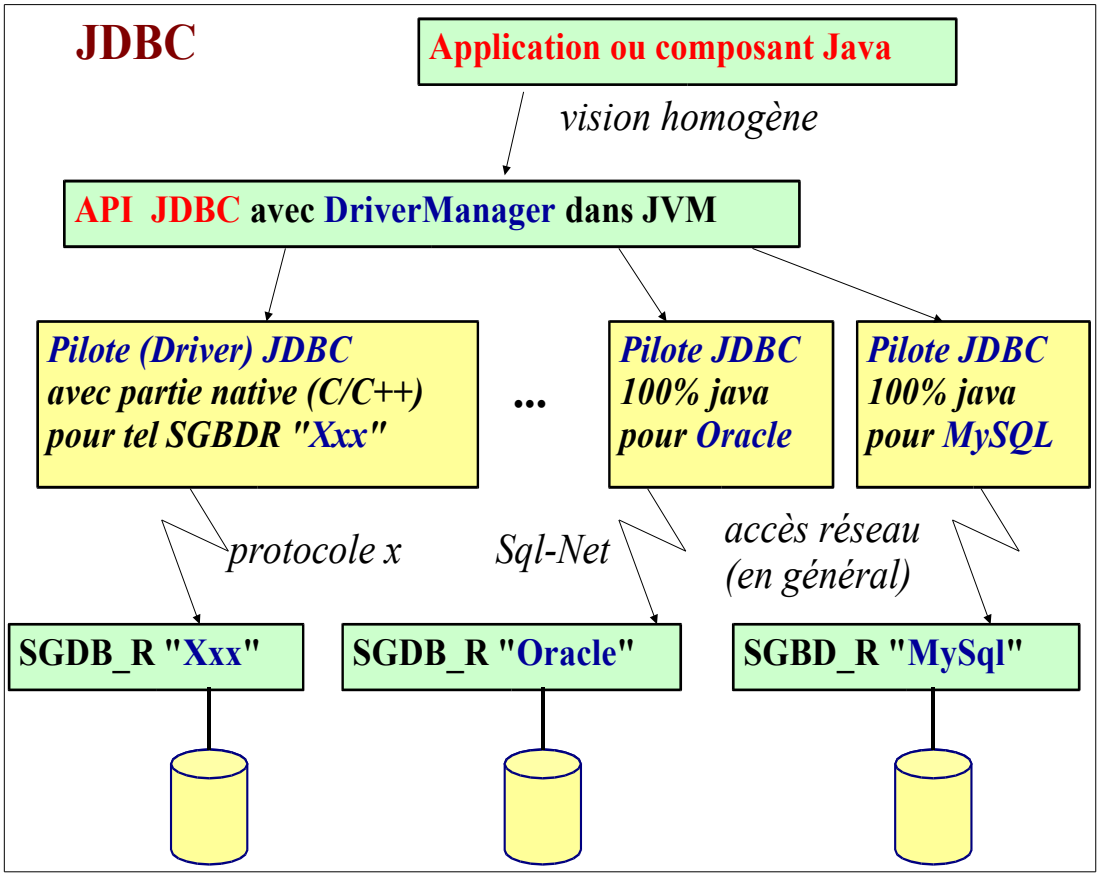
• Il s'agit d'une **API standard** et de bas niveau ***permettant à un programme Java de s'interfacer*** avec **une base de données relationnelle** quelconque (Oracle, DB2, Informix, ...) .

• L'api **JDBC** correspond au package **java.sql** .

• ***javax.sql.DataSource*** est une extension standard permettant de gérer les ***Pools de connexions***.

• Principales fonctionnalités de l'api JDBC :

* Effectuer une connexions vers une base de données
* Lancer des ordres SQL "Insert into, Delete From , Update, ..."
* Récupérer des enregistrements suite à une requête "Select ... From"
* Récupérer des informations sur la structure d'une base (MetaData)
* Déclencher des procédures stockées
* ...



La solution idéale est un pilote JDBC 100% Java donc indépendant de toute plate-forme.

Quel que soit le type de driver JDBC , celui-ci est lié à un type de SGBD (Oracle, DB2, Sybase, Informix, ...).
En règle général, les éditeurs de SGBD-R (Oracle , IBM, ...) proposent un télé-chargement gratuit des drivers JDBC nécessaires pour se connecter à leurs produits.

2. Paramétrage et établissement d'une connexion

Paramétrages à effectuer avec JDBC

* Le ***CLASSPATH*** doit comporter le **".jar" ou ".zip"** contenant le code Java du Driver JDBC que l'on souhaite utiliser.

* ***Choix du Driver JDBC à charger en mémoire*** :

Class.forName(*chNomClasseDuDriver*);

"com.mysql.jdbc.Driver" pour ***MySql***
"oracle.jdbc.driver.OracleDriver" pour ***Oracle 8i***

* ***Url désignant la base de donnée vers laquelle on souhaite établir une connexion*** :

Connection cn = **DriverManager.getConnection**(
chUrlDB, [userName],[password]);

"jdbc:mysql://xxxhost/xxxxdb" pour ***MySql***
"jdbc:oracle:thin:@xxxhost:1521:myDB" pour ***Oracle 8i***

SGBD-R	*archive contenant le code java du friver JDBC*	*classe du pilote JDBC*	*URL*
Oracle	...\ora81\jdbc\lib\classe12.zip	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@localhost:1521:myDB
DB2	...\SQLLIB\java\db2java.zip		
MySQL	...\mysql\lib\mysql-connector-java-3.0.16-stable-bin.jar	com.mysql.jdbc.Driver	jdbc:mysql://localhost/xxxxdb
Access	dans runtime JVM Win32	sun.jdbc.odbc.JdbcOdbcDriver	jdbc:odbc:dsnxxx
HsqlDB		org.hsqldb.jdbcDriver	jdbc:hsqldb:c:\...\BankDB

2.1. Connexion via DriverManager.getConnection()

En utilisant simplement le JDK standard (et l'api JDBC incorporée) , n'importe quel programme java peut établir une connexion JDBC de la façon suivante:

```
import java.sql.*;
....
// chargement de la classe de pilote:
//Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // version ODBC
Class.forName("com.mysql.jdbc.Driver"); // version mysql (mysql.jar)

// Création de l'URL identifiant la base de données
// format --> jdbc:nom_du_pilote:nom_source_de_donnees;param1=val1;...
//String  chUrl_1 = "jdbc:odbc:dsn1";
String  chUrl_2 ="jdbc:mysql://localhost/test"; // version MySql

// Ouverture de la connection:
Connection connexion = DriverManager.getConnection(chUrl_2,
                                                   /* "user" */,  /* "passwd" */ );
connexion.close();// Déconnexion (très important ==> dans finally {} )
```

NB: en cas d'erreur (échec au niveau de la connexion) , on se retrouve dans le bloc

```
catch(  SQLException  /* ou Exception */ ex  ) {
...}
```

NB: Pour éviter d'utiliser des paramètres fixés "en dur" dans le code java , on utilise généralement un fichier de propriétés (à placer dans le CLASSPATH):

paramDB.properties

```
#driver JDBC (ex: com.mysql.jdbc.Driver pour MySql)
#(necessite "mysql-connector-...jar dans le CLASSPATH)
driver=sun.jdbc.odbc.JdbcOdbcDriver
#url de la base de donnée (ex: jdbc:mysql://localhost/baseX)
url=jdbc:odbc:Geo
username=
password=
```

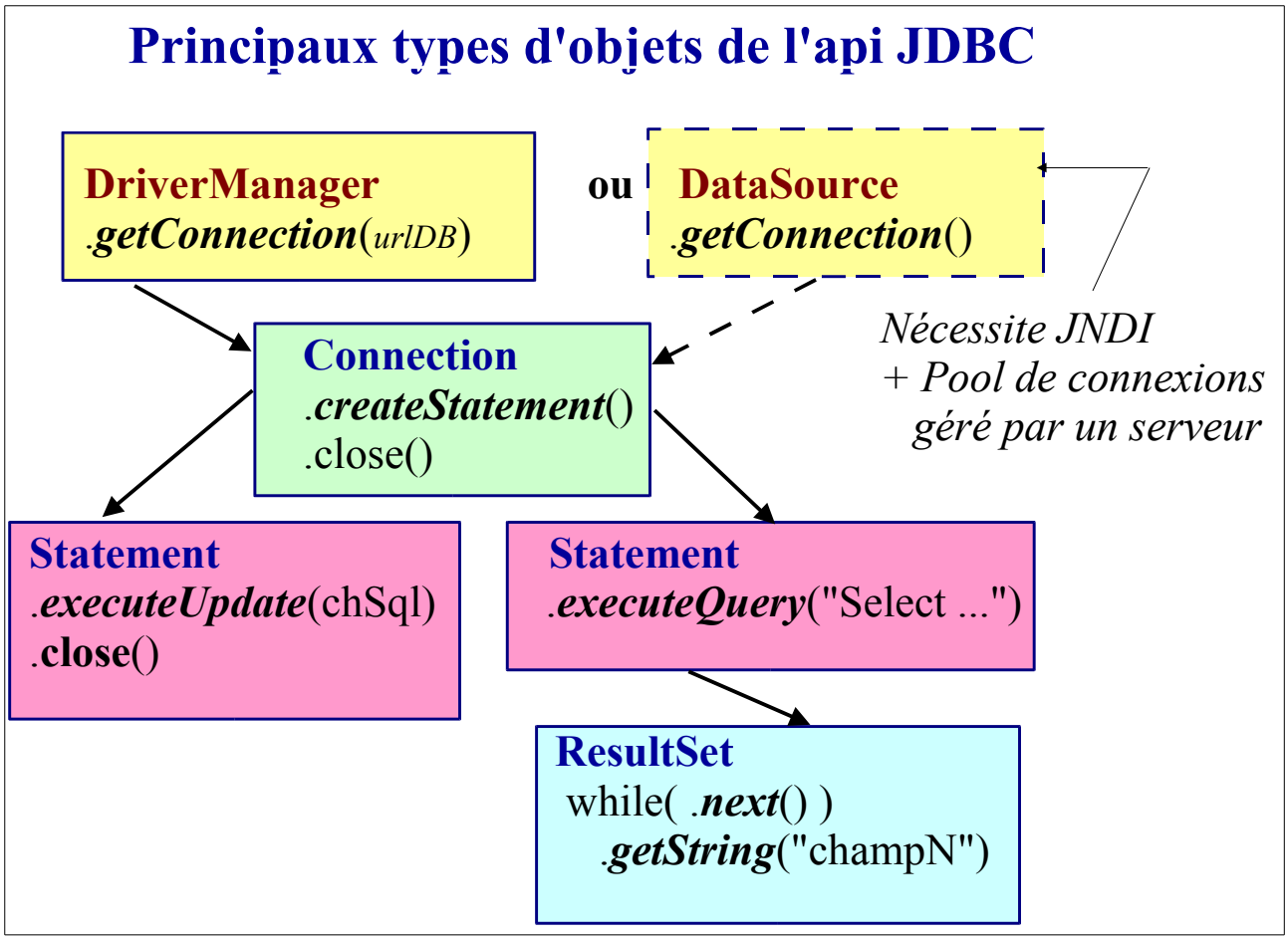
```
ResourceBundle ressources = ResourceBundle.getBundle("paramDB") ; // paramDB.properties
String driver = ressources.getString("driver");  String chUrl = ressources.getString("url");
String username = ressources.getString("username"); String password = ressources.getString("password");
Class.forName(driver);   cn = DriverManager.getConnection(chUrl,username,password) ; ...
```

2.2. via un pool de connexions (DataSource)

Le package additionnel **javax.sql** comporte le type **DataSource** qui correspond à un accès à un **pool de connexions** pris en charge par un serveur d'applications J2EE (ex: Tomcat , WebSphere , JBoss, WebLogic , Jonas , ...).

...

3. Principaux objets de l'api JDBC



4. Lancer un ordre sql

```
Statement statement=connexion.createStatement();

String chOrdreSql = "DELETE FROM telephone WHERE nom='dupond' ";

int nbLignesAffectees= statement.executeUpdate(chOrdreSql);

...

statement.close(); // à ne pas oublier pour ne pas perturber les traitements ultérieurs
```

NB: Ceci doit être englobé au sein d'un traitement d'exceptions : *try/ catch(SQLException)*

5. Effectuer une requête (select)

```
Statement statement=connexion.createStatement();

String chRequeteSql = "SELECT *  FROM telephone WHERE numero LIKE '01%' ";

try {
  ResultSet rs = statement.executeQuery(chRequeteSql);
    }
catch( SQLException ex) { ... }
```

6. Balayer les lignes du résultat

```
while(rs.next())
  {
  String nom = rs.getString("NOM"); // récupérer la valeur du champ "NOM"
  String numero = rs.getString("NUMERO");  ...
  }
rs.close();  // à ne pas oublier !!!
```

NB:

- suivant le type du champ , on utilisera une des fonctions **getXxx**() disponible dans la classe **ResultSet .**
- La fonction **getObject**() permet de récupérer la valeur d'un champ de type quelconque sous la forme d'un objet (classe **Integer**, **Double**, **String** , ...) sur lequel on peut toujours invoquer la fonction **toString**().
- Si la valeur d'un champ n'est pas renseignée , sa valeur nulle (null sql) sera vue comme une valeur en retour de type 0 ou null(JAVA).
  D'autre part, la méthode booléenne **wasNull**() permet de savoir si la valeur d'un champ est le null sql.
- Les versions 2, 3 et 4 de JDBC offre un balayage bi-directionnel via les méthodes **first**() , **last**() , **previous**() , **absolute**(pos) , **relative**(offset) . De plus les méthodes **isFirst**(), **isLast**(), **isBeforeFirst**() et **isAfterLast**() permettent de tester la position courante.

7. Accès à la structure de la base (MetaData)

L'api JDBC permet de s'enquérir de la structure d'une base quelconque.
On peut récupérer la liste des tables d'une base et la liste des champs d'une certaine table:

```
DatabaseMetaData meta = cn.getMetaData();              // cn = objet de type Connection
 System.out.println("\n Liste des tables de la base:");
 String tabOfTableType[] = {"TABLE","VIEW"};
ResultSet rs = meta.getTables(null,null,"%",tabOfTableType);
 while( rs.next())
    {  System.out.print("Nom: "+ rs.getObject(3) );
       System.out.println(" - Type: "  + rs.getObject(4));
    }
 rs.close(); // indispensable
```

```
System.out.println("\n Liste des champs de la table TableAdr:");
   rs = meta.getColumns(null,null,"tableAdr","%");
   while( rs.next())
              System.out.println(rs.getObject(4));
   rs.close(); // indispensable
```

```
   System.out.println("\n Liste des procedures :");
   rs = meta.getProcedures(null,null,"%");
   while( rs.next())
            System.out.println( rs.getObject(3));
   rs.close(); // indispensable
```

Soit **rs** le fruit d'une requête SQL, on peut alors connaître la liste des champs grâce aux instructions suivantes:

```
ResultSetMetaData rsMeta = rs.getMetaData();
int nbChamps = rsMeta.getColumnCount();
for(int i=1;i<=nbChamps;i++)
  { String chNomChamp = rsMeta.getColumnLabel(i);
    int dataType = rsMeta.getColumnType(i);
    switch(dataType)
     {
      case Types.VARCHAR:
      case Types.CHAR:
          chValChamp = rs.getString(i); ...  break;
      case Types.INTEGER:
          chValChamp = String.valueOf( rs.getInt(i) ); ...  break;
      ...
   }
```

Attention: Les **numéros des champs** vont de **1 à n** (et pas de 0 à n-1).

8. Gestion des transactions (tout ou rien)

```
connexion.setAutoCommit(false); // true par défaut

// quelques ordres SQL (Mises à jour , Insertions , Suppressions)

if(...)
   connexion.commit(); // pour valider Toutes les màj.
else
  connexion.rollback(); // pour annuler toutes les màj depuis le dernier commit/rollback
```

9. Préparer et lancer n fois un ordre Sql paramétrable

```
PreparedStatement pstmt = cn.prepareStatement( "Insert Into Table2 Values (?,?)" );
  for(int i=0;i<3;i++)
    {
     pstmt.setString(1,"Valeur"+i  /*valeur du champ1 (texte)*/ );
     pstmt.setInt(2,i /*valeur du champ2 (numérique)*/ );
     pstmt.executeUpdate();
    }
 …
```

10. Appels de procédures stockées

```
  CallableStatement cstmt = cn.prepareCall( "{call fromNom(?)}" );

  cstmt.setString(1/*numéro du param*/,"Power User" /*valeur*/);

  rs = cstmt.executeQuery();
  while( rs.next())
     {
     System.out.print( rs.getObject(1) + ",");
     System.out.print( rs.getObject(2)+ ",");
     System.out.print(rs.getObject(3)+ ",");
     System.out.print( rs.getObject(4)+ ",");
     System.out.println( rs.getObject(5));
     }
  rs.close();
```

11. Astuce pour fermer proprement les connexions

```
static void closeCn(Connection cn){
          try { cn.close(); } catch (SQLException e) {e.printStackTrace();}
        }
static void closeSt(Statement st){
          try { st.close(); } catch (SQLException e) {e.printStackTrace();}
        }
static void closeRs(ResultSet rs){
          try { rs.close(); } catch (SQLException e) {e.printStackTrace();}
        }
…
Connection cn = initConnection();
Statement st = null;    ResultSet rs=null;
try { st = cn.createStatement();
        rs = st.executeQuery("select * from Compte");
        while(rs.next()){        ...
        }
} catch (SQLException e) {
                        e.printStackTrace();
} finally{
        closeRs(rs); closeSt(st); closeCn(cn);
}
...
```

12. Récupérer la valeur d'une clef auto-incrémentée

```
public Long insertNewCompte(Compte cpt){
        Long pk=null;  Connection cn = initConnection();
        PreparedStatement pst = null;     ResultSet rsKeys = null;
        try { pst = cn.prepareStatement("insert into Compte(label,solde) values(?,?)");
                pst.setString(1, cpt.getLabel());    pst.setDouble(2, cpt.getSolde());
                pst.executeUpdate(); //avec auto_increment mysql sur colonne numCpt
                rsKeys = pst.getGeneratedKeys();//récupérer valeur clef primaire (ici numCpt)
                if(rsKeys.next()){ pk= rsKeys.getLong(1);
                }
        }catch(SQLException e) { e.printStackTrace();
        } finally{ closeRs(rsKeys);closeSt(pst);closeCn(cn);
        }
        return pk;
}
```

13. Fonctionnalités à partir de la version 2 de JDBC

La version 2 de JDBC (accompagnant le JDK 1.2) , offre les fonctionnalités suivantes:

- Gestion des champs binaires (**blob**) et nouveaux types .
- Gestion de **curseur à plusieurs lignes** (comme ODBC)
- Gestion directe des **mises à jour** depuis l'objet **ResultSet** (méthodes UpdateXXX())
- ...

13.1. Types de champs

La classe **java.sql.Types** comporte les constantes suivantes (types **XOPEN**) :

ARRAY	SQL ARRAY (Depuis JDBC2)
BIGINT	
BINARY	
BIT	
BLOB	Binary Large OBject (ex: image) (Depuis JDBC2)
CHAR	
CLOB	Character Large OBject (ex: memo) (Depuis JDBC2)
DATE	
DECIMAL	
DISTINCT	User Defined Data Type (Depuis JDBC2)
DOUBLE	
FLOAT	
INTEGER	
JAVA_OBJECT	Objet Java (Depuis JDBC2)
LONGVARBINARY	
LONGVARCHAR	
NULL	
NUMERIC	
OTHER	
REAL	
REF	(Depuis JDBC2)
SMALLINT	
STRUCT	User Defined Data Type (Depuis JDBC2)
TIME	
TIMESTAMP	
TINYINT	
VARBINARY	
VARCHAR	

==> Consulter l'aide en ligne pour les détails.

13.2. Mises à jour directes à partir d'un objet ResultSet

13.2.a. Ajout d'un nouvel enregistrement:

```
rs.moveToInsertRow();  // préparer un nouvel enregistrement vierge en mémoire
rs.updateString("nom","dupond");  // donner une valeur au champ 1
...
rs.updateInt("age",30); // donner une valeur au champ n
rs.insertRow(); // ajouter le nouvel enregistrement dans la base
```

13.2.b. Mise à jour de l'enregistrement courant:

```
int age = rs.getInt("age");
```

```
rs.updateInt("age",age+1);  // modifier 1 ou plusieurs champ en mémoire
rs.updateRow();   // enregistrer les modifications dans la base de données.
```

13.2.c. Suppression de l'enregistrement courant:

```
rs.deleteRow();
```

Notes:

Les instructions ci-dessus ne fonctionneront correctement que si toutes les conditions suivantes sont vérifiées:

- Le **driver JDBC est à la hauteur** (supportant ces nouvelles fonctionnalités de la version 2 de JDBC).
- Le ResultSet provient d'une **requête SQL simple** (une seule table , pas de jointure).
- L'objet Statement qui a permis d'effectuer la requête à été ouvert en passant 2 paramètres à la fonction cn.**CreateStatement**(- , -);

14. Mémento SQL + Mise en oeuvre MySQL

14.1. Exemples (simples) de requêtes SQL

```
SELECT fieldlist        FROM tablenames                  [WHERE    searchcondition]
 [GROUP BY fieldlist   [HAVING searchconditions] ]    [ ORDER BY fieldlist ]
```

- **SELECT DISTINCT** Last_Name **FROM** Employees **WHERE** Last_Name = 'Smith'
- **SELECT Count**(*), **Avg**(Salary) , **Max**(Salary) **FROM** Employees
- **SELECT** Department, **Count**(Department) **FROM** Employees **GROUP BY** Department **HAVING Count**(Department) > 100

- **DELETE FROM** Employees **WHERE** Title = 'Trainee'

- **UPDATE** Orders **SET** Freight = Freight * 1.03 **WHERE** Ship_Country = 'UK'
- **UPDATE** Orders **SET** Amount = Amount * 1.1, Freight = Freight * 1.03 **WHERE** Country = 'UK'

- **INSERT INTO** T_Rubrique (id,label) **VALUES** (2 , "Automobile")

14.2. Exemples de scripts pour créer une base "MySQL"

1. ***télécharger le serveur MySQL depuis l'url*** ***http://dev.mysql.com/***
1. ***installer le logiciel***
2. ***lancer MySQL/bin/WinMySQLAdmin.exe ou MySQL/bin/MySQLInstanceConfig.exe pour paramétrer le serveur et faire en sorte qu'il puisse démarrer comme un service***
3. ***Lancer ensuite les scripts suivants:***

set_env_mysql.bat

```
set MYSQL_HOME=C:\Prog\DB\MySQL\MySQL_Server_4.1
set MYSQL_BIN=%MYSQL_HOME%\bin
```

1_lancer_pwd_root.bat

```
call set_env_mysql.bat
```

```
REM  Fixer le mot de passe de l'administrateur "root" de mysql (ex: "root")
%MYSQL_BIN%\mysql -h localhost -u root -p < update_root_user_with_root_pwd.txt
pause
```

update_root_user_with_root_pwd.txt

```
USE mysql;
UPDATE user SET Password=PASSWORD('root')  WHERE user='root';          FLUSH PRIVILEGES;
```

2_lancer_delete_NoPassword.bat

```
call set_env_mysql.bat
%MYSQL_BIN%\mysql -h localhost -u root -p < delete_no_password.txt
pause
```

delete_no_password.txt

```
USE mysql;       DELETE FROM user WHERE User='';        FLUSH PRIVILEGES;
```

3a_create_devisedb.bat

```
call set_env_mysql.bat
%MYSQL_BIN%\mysql -h localhost -u root -p < create_devisedb.txt
pause
```

create_devisedb.txt

```
#DROP DATABASE devisedb;
CREATE DATABASE devisedb;
USE devisedb;
CREATE TABLE DEVISE(MONNAIE VARCHAR(64) NOT NULL PRIMARY KEY,DCHANGE DOUBLE);

INSERT INTO DEVISE VALUES('Dollar',1.0);     INSERT INTO DEVISE VALUES('Euro',1.05);
INSERT INTO DEVISE VALUES('Livre',0.7);      INSERT INTO DEVISE VALUES('Yen',2.1);
show tables;
```

4a_lancer_grant_priv_devisedb.bat

```
call set_env_mysql.bat
%MYSQL_BIN%\mysql -h localhost -u root -p < grant_priv_on_devisedb.txt
pause
```

grant_priv_on_devisedb.txt

```
# GRANT ALL PRIVILEGES
  GRANT SELECT,INSERT,UPDATE,DELETE
     ON devisedb.*
       TO mydbuser@'%'
         IDENTIFIED BY 'mypwd';
FLUSH PRIVILEGES;
```

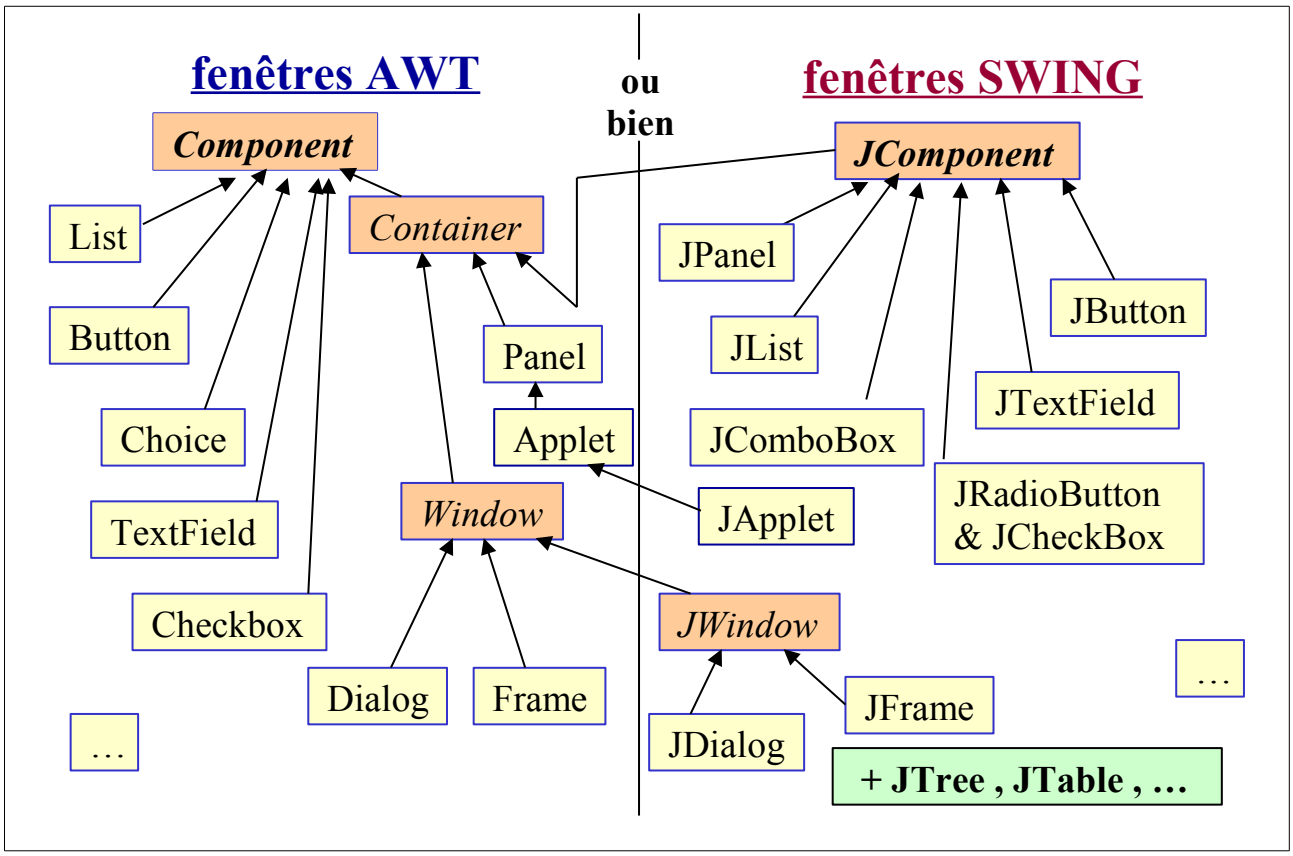
X - Api pour IHM/GUI (swing, ...)

1. Eléments de base sur AWT/SWING et événements

1.1. Gestion des fenêtres et des contrôles

Pour gérer les fenêtres , il existe (au choix) deux api (relativement proches et parallèles):

- **Les classes (de java.awt) héritant de Component**. Il s'agit d'une encapsulation JAVA des api natives du système d'exploitation.
- **Les classes (de javax.swing) héritant de JComponent** (fenêtre 100 % java).



1.1.a. Composant de l'interface graphique (Component)

La classe abstraite java.awt.**Component** hérite directement de **Object** et correspond à un composant quelconque de l'interface graphique au sens large (fenêtre, contrôle,…). Parmi les méthodes de cette classe on retiendra principalement les suivantes:

getParent()	retourne le Container contenant l'objet
setBackground(color)	Fixe la couleur de fond
setForeground(color)	Fixe la couleur d'avant plan (texte,traits,…)
setFont(fonte)	Fixe la police de caractères
setVisible(booléen)	cache ou rend visible l'objet graphique
setEnabled(booléen)	active ou inactive (grise) l'objet
setLocation(x,y), **setSize**(w,h)	déplace , redimensionne l'objet
requestFocus()	demande le focus sur cet objet

bounds()	retourne la position et dimension de l'objet sous la forme d'un objet Rectangle (coordonnée)

Dans l'arbre d'héritage on trouvera directement sous la classe Component une série de classes correspondant aux contrôles (boutons poussoirs, zones d'éditions , …) plus une classe abstraite nommée java.awt.Container .

1.1.b. Fenêtres "Container" et sous fenêtres

La classe java.awt.**Container** correspond à un Conteneur (fenêtre pouvant contenir d'autres sous fenêtres). Ses principales méthodes sont les suivantes:

add(component)	Ajoute un composant (contrôle,panel, …)
remove(component)	Enlève un sous objet
countComponents()	Retourne le nombre de composants
setLayout(layoutMgr)	Associe un gestionnaire de répartition au conteneur
locate(x,y)	Retourne une référence sur le Component en (x,y)

- Les classes java.awt.**Panel** et javax.swing.**JPanel** correspondent à un "**sous-paneau**" . Il s'agit d'un élément généralement invisible qui permet de gérer finement la disposition des contrôles dans une fenêtre en jouant le rôle de **Conteneur intermédiaire**: un objet Panel est toujours contenu dans un autre conteneur et contient à son tour d'autres contrôles ou sous conteneurs.

- Les classes "Fenêtre" (java.awt.**Window** et javax.swing.**JWindow**) correspond à une fenêtre applicative (sans bordure , ni menu) qui comporte deux sous classes importantes: java.awt.*Dialog* (boîte de dialogue) et java.awt.*Frame* (fenêtre cadre ou fenêtre principale).

- Les classes java.awt.**Frame** et javax.swing.**JFrame** correspondent à une fenêtre principale qui peut avoir un titre, une barre de menu, une bordure permettant de la redimensionner , un icône et un curseur spécifique. Cette classe est très souvent sous classée car elle correspond au point central d'une application graphique. Les principales méthodes de **Frame** sont:

setTitle(titre)	Change le titre
setMenuBar(barreDeMenu)	Associe un menu à la fenêtre
setResizeable(monBooleen)	true / false

1.1.c. Les Contrôles (composants graphiques élémentaires)

Libellé (étiquette)

```
lbl = new javax.swing.JLabel("Nom :");
```

Champ de saisie simple (une seule ligne)

```
champ = new javax.swing.JTextField();
champ.setText("Valeur par défaut");
String  texteSaisi = champ.getText();
champ.select(2 /*start*/ ,6 /*end*/);
```

```
String texteSelectionne = champ.getSelectedText();
```

Bouton poussoir

```
jbtnOk = new javax.swing.JButton("Ok");
jbtnOk.setText("OK");
```

"Case à cocher" ou "bouton radio" accompagné de son libellé.

En version swing:

2 classes différentes:

- **JCheckBox** pour les cases à cocher (non exclusives).
- **JRadioButton** pour les boutons "radio" (exclusifs entre eux).

Paramétrage de l'exclusivité entre différentes options:

```
ButtonGroup bg = new ButtonGroup();  // objet invisible
bg.add(jRadioButtonMarie);
bg.add(jRadioButtonCelibataire);
```

Liste déroulante pour choisir un élément parmi n.

En version swing:

```
combo = new javax.swing.JComboBox();
combo.addItem("rouge");
combo.addItem("vert"); combo.addItem("bleu");

combo.setSelectedItem("vert"); //ou combo.setSelectedIndex(1);

nbElts = combo.getItemCount();
int selIndex = combo.getSelectedIndex();
String chaineChoisie = combo.getSelectedItem();
```

Liste d'éléments (avec sélection multiple possible)

En version swing:

```
Vector vectSel = new Vector();
JList jListSel = new JList();
...
vectSel.addElement(uneChose);
...
...
jListSel.setListData(vectSel);

...
```

```
int tabSelIndex[] = jListSel.getSelectedIndices();
```

Zone de saisie multi-lignes

En version swing:

classe	*fonctionnalités*
JTextArea	zone de texte simple à plusieurs lignes (light).
JTextPane	éditeur de texte sophistiqué (mise en forme possible des caractères).
JEditorPane	Zone de texte sophistiquée (text/plain ou text/html)

Boite de dialogue

- javax.swing.**JDialog** est la classe générique des boites de dialogue en version *SWING*.
- Un choix de nom de fichier s'effectue avec la classe **JFileChooser** :

```
JFileChooser chooser = new JFileChooser();

int returnVal = chooser.showOpenDialog(parent);
if(returnVal == JFileChooser.APPROVE_OPTION)
  {
  fileName= chooser.getSelectedFile().getName();
  }
```

Menu

JMenuBar, JMenu, JMenuItem *(à imbriquer et à attacher à la fenêtre principale)*
JPopupMenu ==> menu contextuel (sur click droit)

1.1.d. Fonctionnalités générales des composants SWING élémentaires:

.**setToolTipText**("Chaine InfoBulle")	précise le texte de la bulle d'aide
.**setVisible**(true/flase)	montre ou cache le composant
.**setEnable**(false/true)	grise ou dégrise le composant
.**setOpaque**(false)	précise la "non transparence" du fond

Nb: ces méthodes proviennent de **JComponent**

1.1.e. Gestion simple du scrolling (JScrollPane)

```
JScrollPane jScrollPaneXXX = new JScrollPane();
...
jScrollPaneXXX.getViewport().setView(jUneChose); // chose = liste, panel ,arbre, ...
```

1.1.f. Boîte de message et Prompt

La classe **JOptionPane** comporte quelques **méthodes statiques** qu'il suffit d'appeler directement (en préfixant par le nom de la classe) et qui permettent d'afficher des boîtes de messages et des invites pour saisir des valeurs:
(*NB*: **p=this** ou **null** ou **fenêtre parente**)

méthode statique	*fonctionnalité*
JOptionPane**. showMessageDialog**(p,"Bienvenue");	Affiche une boîte de message
name= JoptionPane.**showInputDialog**(p,"Quel est votre nom");	Prompt
JoptionPane.**showConfirmDialog**(p,"voulez vous ...?") => renvoie une valeur du genre OK_OPTION ou YES_OPTION	Demande de confirmation (Yes,No,Cancel)

Paramètres optionels de ces méthodes statiques:

title : titre de la mini boîte de dialogue.
optionType: YES_NO_OPTION ou OK_CANCEL_OPTION ou ...
messageType: WARNING_MESSAGE ou ERROR_MESSAGE ou QUESTION_MESSAGE ou INFORMATION_MESSAGE.

1.1.g. Onglets (JTabbedPane)

La classe **JTabbedPane** correspond à un conteneur de type "Série d'onglets".
Chaque onglet sera programmé sous la forme d'un élément héritant de **JPanel** .

Le simple fait d'incorporer plusieurs panneaux (JPanel) dans un conteneur de type "JTabbedPane" permet d'obtenir le fameux Look à onglets.

Chaque onglet est associé à un nom ou libellé (sous forme de chaîne de caractères):

```
JTabbedPane jSerieOnglets = new JTabbedPane();
...
jSerieOnglets.addTab("Onglet1", jPanelOnglet1);
jSerieOnglets.addTab("Onglet2", jPanelOnglet2);
```

1.2. Gestionnaire de répartition (LayoutManager)

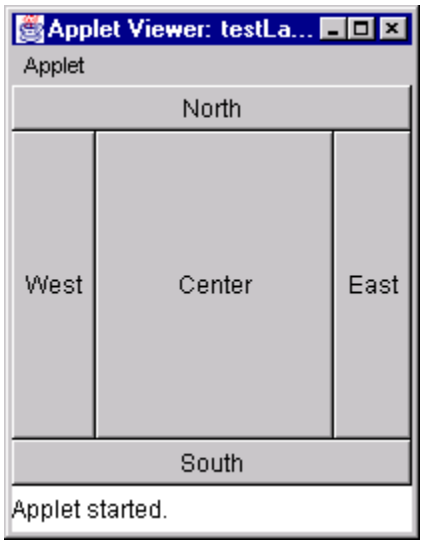
Un gestionnaire de répartition (défini par l'interface ***LayoutManager***) est un **objet invisible** qui sera **associé à un conteneur** et qui servira à disposer au mieux les contrôles qui seront ultérieurement placés dans le conteneur.

Le programmeur a simplement besoin d'installer (de façon facultative) le gestionnaire de répartition.
→ conteneur.**setLayout**(layoutManager);
Celui-ci sera alors automatiquement utilisé par la suite.

Il faudra tout de même préciser certains paramètres lors de l'ajout d'un contrôle dans le conteneur→exemple: conteneur.**add**(***"South",***contrôle);

JAVA dispose de quelques gestionnaires de répartition prédéfinis:

BorderLayout	***Arrange les contrôles sur les bords et au centre du conteneur*** en fonction des indications "**South**", "**North**", "**East**", "**West**", "**Center**" . Les paramètres facultatifs *hgap* et *vgap* du contructeur permettent de préciser l'espacement entre deux composants adjacents.
FlowLayout *(par défaut)*	Arrange les composants en ligne de gauche à droite. Fait entrer autant de composants que possible sur la ligne courante avant de passer à la prochaine. Au sein d'une ligne les choses sont alignées en fonction du paramètre facultatif align du constructeur. FlowLayout.CENTER , .LEFT , .RIGHT
GridLayout, CardLayout, GridBagLayout, ...	... autres gestionnaires (complexes)...



1.2.a. Positionnement absolu (sans LayoutManager)

this./*getContentPane().*/**setLayout(null);**

jTextField1.**setBounds(new Rectangle(126, 8, 63, 21));**

1.3. Classe Graphics – pour Dessiner (lignes, rectangles, ...)

Pour dessiner quoi que ce soit dans une fenêtre ou dans un autre périphérique graphique (imprimante, mémoire) , il faut passer par la classe Graphics.
Cette classe est à mettre en parallèle avec la notion de "Graphic Context (gc)" de X-Window ou bien encore avec le "Device Context (DC)" de Win32.

La classe java.awt.**Graphics** permet de préciser comment les chose seront dessinées (Coordonnées , fontes , couleurs, modes des tracés, styles des hachures, …).
C'est aussi au travers d'elle que seront invoquées les méthodes permettant de dessiner telles que .drawLine() ou fillRect().

Dans la pratique, on ne peut pas directement créer une nouvelle instance de la classe Graphics qui est d'ailleurs abstraite mais on procédera de la façon suivante:

- En manipulant l'instance "graphics" qui nous est fournie par la méthode **paint**() permettant de (re)dessiner le contenu de la fenêtre qui a besoin d'être rafraîchie.

- En obtenant l'instance en invoquant la méthode **getGraphics**() des classes Component ou Image.

```
public abstract class java.awt.Graphics extends java.lang.Object  {
      // Constructors:  protected Graphics();
   public abstract void clearRect(int  x, int  y, int width, int  height);
   public abstract void clipRect(int  x, int  y,  int width, int height);
   public abstract void drawArc(int  x, int  y, int  width, int  height, int  startAngle, int  arcAngle);
   public abstract boolean  drawImage(Image  img, int  x, int  y, ImageObserver  observer);
   ...
   public abstract void drawLine(int  x1, int  y1, int  x2,  int  y2);
   public abstract void drawOval(int  x, int y, int width, int  height);
   public abstract void  drawPolygon(int  xPoints[], int yPoints[],int  nPoints);
   public void drawRect(int  x, int  y, int  width, int  height);
   public abstract void drawRoundRect(int  x, int  y, int  width, int  height,
                                      int  arcWidth, int  arcHeight);
   public abstract void  drawString(String  str, int  x, int  y);
   public abstract void  fillArc(int  x, int  y, int  width, int  height, int  startAngle, int  arcAngle);
   public abstract void fillOval(int  x, int  y, int  width, int  height);
   public abstract void  fillPolygon(int  xPoints[], int  yPoints[], int  nPoints);
   public abstract void fillRect(int  x, int  y, int  width,  int  height);
   public abstract void fillRoundRect(int  x, int  y, int  width, int height,
                                      int  arcWidth,  int  arcHeight);
   ...
   public abstract void setColor(Color  c);
   public abstract void setFont(Font  font);
   public abstract void setPaintMode();
   public abstract void setXORMode(Color  c1);
   public abstract void   translate(int  x, int  y);
}
```

2. Gestion des événements

Le principe de fonctionnement du modèle événementiel en vigueur depuis le jdk 1.1 **est basé sur le modèle "fournisseur / abonnés":**
L'événement sera envoyé à tous les objets qui auront préalablement marqué leurs intérêts vis à vis de celui-ci via une procédure d'enregistrement.

Au sein de ce modèle événementiel, il faut considérer plusieurs entités :

- L'objet qui génère l'événement (fenêtre, contrôle, …) (***source d'événement***)
- Un (ou plusieurs) objet(s) qui vont traiter (gérer) l'événement (***listener***).
  + un éventuel objet *intermédiaire* appelé ***adaptateur***
- Un objet (**event**) qui va véhiculer les détails (données) de l'événement.

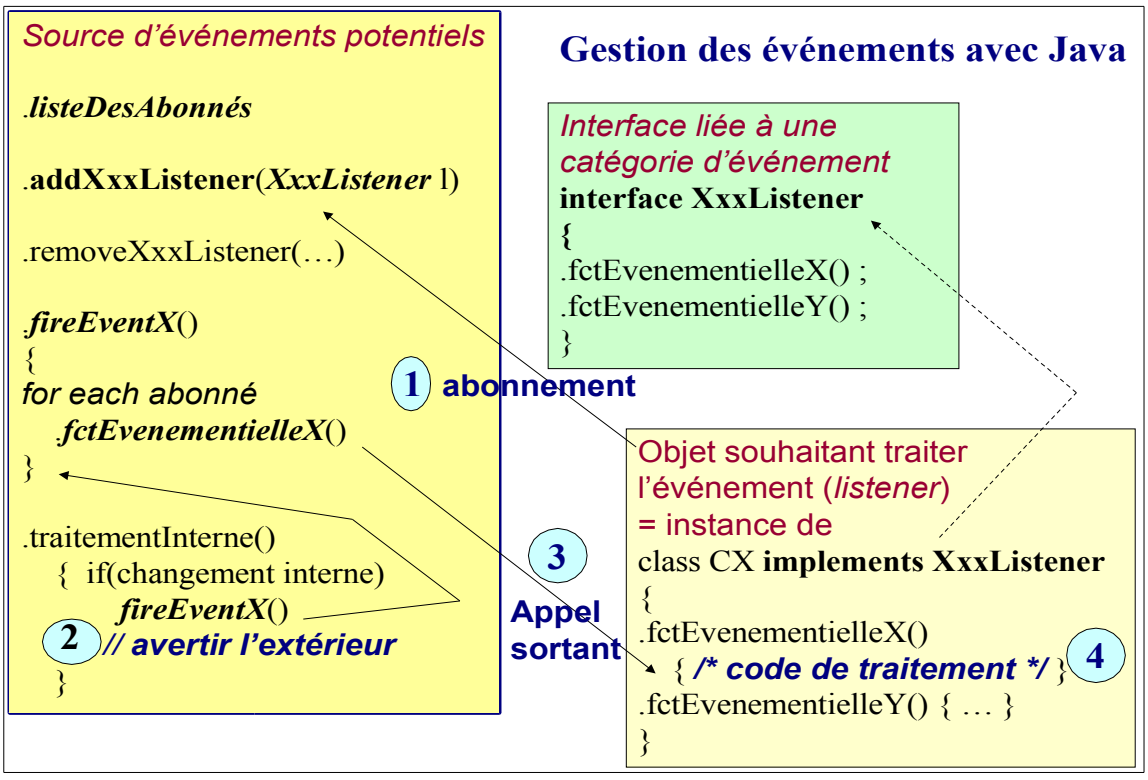
Dans le cas le plus simple une fenêtre parente (englobante) pourra gérer le rôle de listener.

2.1.a. Notion d'abonnement

Le fait de s'abonner se code de la façon suivante:
composantSource.**add*XXX*Listener**(objetQuiSouhaiteGererEvenement);
composantSource.**add*XXX*Listener**(deuxièmeAbonné);
>>>> ***XXX*** *correspond içi à une* ***catégorie d'événements***. <<<<

Pour chacune de ces catégories (XXX) , il existe:

- une méthode pour s'abonner: ***addXXXListener***()
- une **interface *XXXListener*** que doit implémenter l'objet qui s'abonne et qui souhaite traiter l'événement.
- une sous classe d'événement ***XXXEvent*** héritant de la classe abstraite ***java.awt.AWTEvent*** dérivant elle même de ***java.util.EventObject***.



Exemple:
Si l'on veut traiter au niveau d'une instance de la classe Delegue le click sur un bouton poussoir, il faudra alors coder les choses de la façon suivante:

```
import java.awt.*;  import javax.swing.*;
import java.awt.event.*;  // pour accéder à l'interface XXXListener

class PetiteFenetre extends JFrame{
        public PetiteFenetre()
                {
                Delegue unDelegue = new Delegue();
                JButton monBouton = new JButton("Touch Me");
                monBouton.addActionListener(unDelegue); // abonnement
                this.getContentPane().add(monBouton);
                }
        public static void main(String [] argv)
                {
                JFrame f = new PetiteFenetre();
                f.pack();
                f.setVisible(true);
                }
}
```

```
class Delegue implements ActionListener
{
        ...     /* il faut coder toutes les méthodes de l'interface ActionListener
                → soit une seule méthode içi (cas le plus simple) */
        public void actionPerformed(ActionEvent e)
                {
                // traitement (gestion) du message:
                System.exit(0);
                }
}
```

Dans la plupart des interfaces graphiques(IHM) , l'entité délégué qui traite l'événement est très fortement lié à l'objet fenêtre , Panneau, ou boîte de dialogue qui contient directement le composant source qui sera à la source de l'événement (ex: Bouton poussoir).

NB1: Un composant graphique peut très bien gérer lui même l'événement reçu:
(implements XXXListener et addXXXListener(this);)

NB2: Si c'est la fenêtre parent (qui s'abonne et) qui gère les événements provenant de ses fenêtres filles, il faudra alors distinguer celles-ci via la source de l'événement:

```
Object origine=e.getSource();
        if(origine==sousComposant1)
        {  ...  }
        else if(origine== sousComposant2) ...
```

La plupart des générateurs de code ont généralement recours à l'utilisation des classes imbriquées:

```
class MaFenetre  ....   // classe "conteneur graphique"
{
...
public void  initialiser()
 {  ....
 jButtonOK.addActionListener(new
    /*début du code de la classe imbriquée*/ java.awt.event.ActionListener()
    {
     public void actionPerformed(ActionEvent e)
     {
      jButtonOK_actionPerformed(e);
     } /* fin du code de la classe imbriquée */
    });
  ....
  }
....
void jButtonOK_actionPerformed(ActionEvent e)
  { ... /* traitement de l'événement */ }
}
```

Interfaces liées aux catégories de messages (ayant chacune une méthode d'abonnement addXXXListener et un type d'événement XXXEvent qui leur correspondent) :

Interface / catégorie d'événements	*types de composants sources*
ActionListener	Button (éventuellement JRadioButton, JCheckBox, ...)
MouseListener	Panel, Canvas , ...(éventuellement JList, ...)
KeyListener	Zone de saisie (==> codez en priorité *keyReleased*)
ItemListener	Boutons radios , cases à cocher , zones de listes, ...
...	...

Interface	*méthodes à implémenter (void)*
ActionListener	**actionPerformed**(ActionEvent)
AdjustmentListener	ajustmentValueChanged(AjustmentEvent)
ComponentListener	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) **componentResized**(ComponentEvent)
ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
ItemListener	**itemStateChanged**(ItemEvent)
KeyListener	keyPressed(KeyEvent) **keyReleased**(KeyEvent) keyTyped(KeyEvent)

MouseListener	**mouseClicked**(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) **mousePressed**(MouseEvent) **mouseReleased**(MouseEvent)
MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
TextListener	textValueChanged(TextEvent)
WindowListener	windowActivated(WindowEvent) **windowClosed**(WindowEvent) windowClosing(WindowEvent) windowDesactivated(WindowEvent) windowDeiconifiedWindowEvent) windowIconified(WindowEvent) windowOpened(WindowEvent)

Exemple très classique (nouvelle sélection dans une liste déroulante):

```
jComboBoxTypeBase.addItemListener(new java.awt.event.ItemListener()
    {
     public void itemStateChanged(ItemEvent e)      {
       jComboBoxTypeBase_itemStateChanged(e);
     }
    });
...
void jComboBoxTypeBase_itemStateChanged(ItemEvent e)  {
   int index=jComboBoxTypeBase.getSelectedIndex();
  ...   }
...
```

2.1.b. Adaptateurs

NB: L'API de JAVA fournit des ***adaptateurs par défaut*** qui ne font rien (code vide) sous la forme de ***classes abstraites*** (**MouseAdapter, KeyAdapter, WindowAdapter**, …).
Il suffit alors de dériver une ces classes pour ensuite avoir la possibilité de ne coder (redéfinir) qu'une seule des méthodes de l'interface XXXListener correspondante:

```
class MonAdaptateur extends MouseAdapter {
        public void mouseEntered(MouseEvent e) { … }
}
```

Exemples très classiques:

```
jTreeDep.addMouseListener(new java.awt.event.MouseAdapter()    {
     public void mouseClicked(MouseEvent e)      {
       jTreeDep_mouseClicked(e);
     } });
...
```

```
jTextFieldAnneeMini.addKeyListener(new java.awt.event.KeyAdapter()    {
     public void keyReleased(KeyEvent e)      {
       jTextFieldAnneeMini_keyReleased(e);
     } });
...
```

ANNEXES

XI - Annexe – Tests unitaires (JUnit 3 et 4)

1. Tests unitaires avec JUnit (3 ou 4)

1.1. Présentation de JUnit

JUnit est un ***framework*** simple permettant d'effectuer des **tests** (unitaires , de non régression, ...) au cours d'un développement java . [**Projet Open source ---> http://junit.sourceforge.net/ , http://junit.org]** . ***JUnit est intégré au sein de l'IDE Eclipse*** .

RelaxNG est une technologie de test unitaire concurrente. **JUnit** existe en **versions 3 et 4**.

La **version 4** utilise des **annotations** pour son paramétrage (**@Test** , **@Before** , ...)

Remarque très importante : **La technologie JUnit (3 ou 4) créer automatiquement une instance de la classe de test pour chaque méthode de test à déclencher** → constructeurs , setUp() et méthodes préfixées par @Before seront donc potentiellement appelés plusieurs fois !!!!

1.2. Structure d'une classe de test

Test **Junit 3**:

```
package com.mycompany.app1;
import junit.framework.Test;    import junit.framework.TestCase;

public class CalculateurTest    extends TestCase {
   private Calculateur c;

   protected void setUp(){
     c = new Calculateur();
   }

   public void testAdd()  {
      assertEquals( c.add(5,6) , 11 , 0.000001 );  // ou assertTrue(condition) .
   }

   public void testMult() {
      assertEquals( c.mult(5,6) , 30 , 0.000001 );
   }
}
```

JUnit 3 est basée sur des conventions de nommage:

- Au sein d'une classe de Test JUnit3, la méthode ***setUp***() sera appelée automatiquement pour initialiser les valeurs de certains objets qui seront ultérieurement utilisés au sein des tests. [NB : setUp() sera peut être appelée plusieurs fois : avant chaque test].
- Chaque test correspond à une méthode de type "***testXxx()***" ne retournant rien (**void**) mais effectuant quelques assertions (***Assert.assertXxxx(....)***)
- On peut éventuellement programmer une méthode ***tearDown***() qui sera alors appelée

après chaque terminé (ex: pour ré-initialiser le contenu d'une base après).

Test **Junit 4**:

```
package package com.mycompany.app1;

import org.junit.Assert;
import org.junit.Test;
import org.junit.Before;

/** Unit test for simple Calculateur. (JUnit 4 with annotations) */
public class CalculateurTest
{
   private Calculateur c;

   @Before
   public void initialisation(){
     // déclenché avant chaque @Test .
   }

   /* constructeur par défaut*/
   public  CalculateurTest(){
    c = new Calculateur();
   }

   @Test
   public void testerAdd()  {
      Assert.assertEquals( c.add(5,6) , 11 , 0.000001 ); //ou Assert.assertTrue(5+6==11);
   }

   @Test
   public void testerMult() {
      Assert.assertEquals( c.mult(5,6) , 30 , 0.000001 );
   }
}
```

Il existe @Before , @After (potentiellement déclenchés plusieurs fois [avant/après chaque test]) et @BeforeClass , @AfterClass (pour initialiser des choses "static")

NB: il faut que **JUnit-4...jar** soit dans le classpath /

La démarche conseillée consiste à

*** coder un embryon des classes à programmer (code incomplet)**

*** coder les tests (voir précédemment) et les déclencher une première fois (==> échecs normaux)**

*** programmer les traitements prévus**

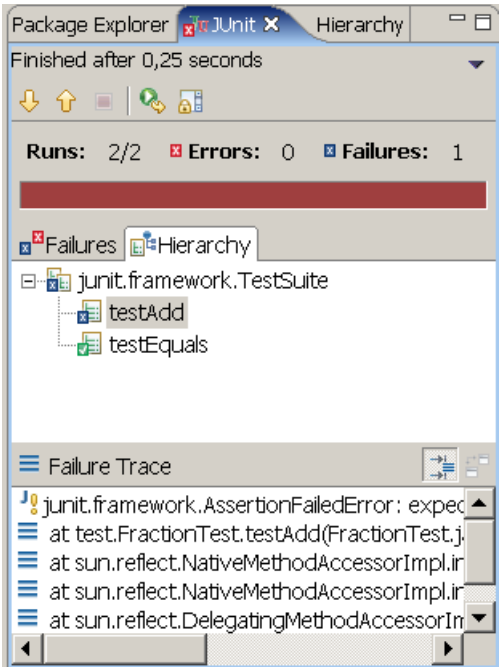
*** ré-effectuer les tests (==> réussite ???)**

*** améliorer (peaufiner) le code**

*** ré-effectuer les tests (==> non régression ???) * ...**

1.3. Lancement des tests unitaires

Depuis eclipse : **Run as / Junit Test**



***Comptabilisations*:**

Error(s) : exceptions java non rattrapées.
Failure(s) : assertions non vérifiées.
VERT si aucune erreur.

Depuis maven :

coder les tests unitaires dans **src/test/java**
et lancement via **mvn test**

Remarques importantes:

- Il vaut mieux s'habituer à la version 4 de JUnit car elle est utilisée par les versions récentes de Spring.

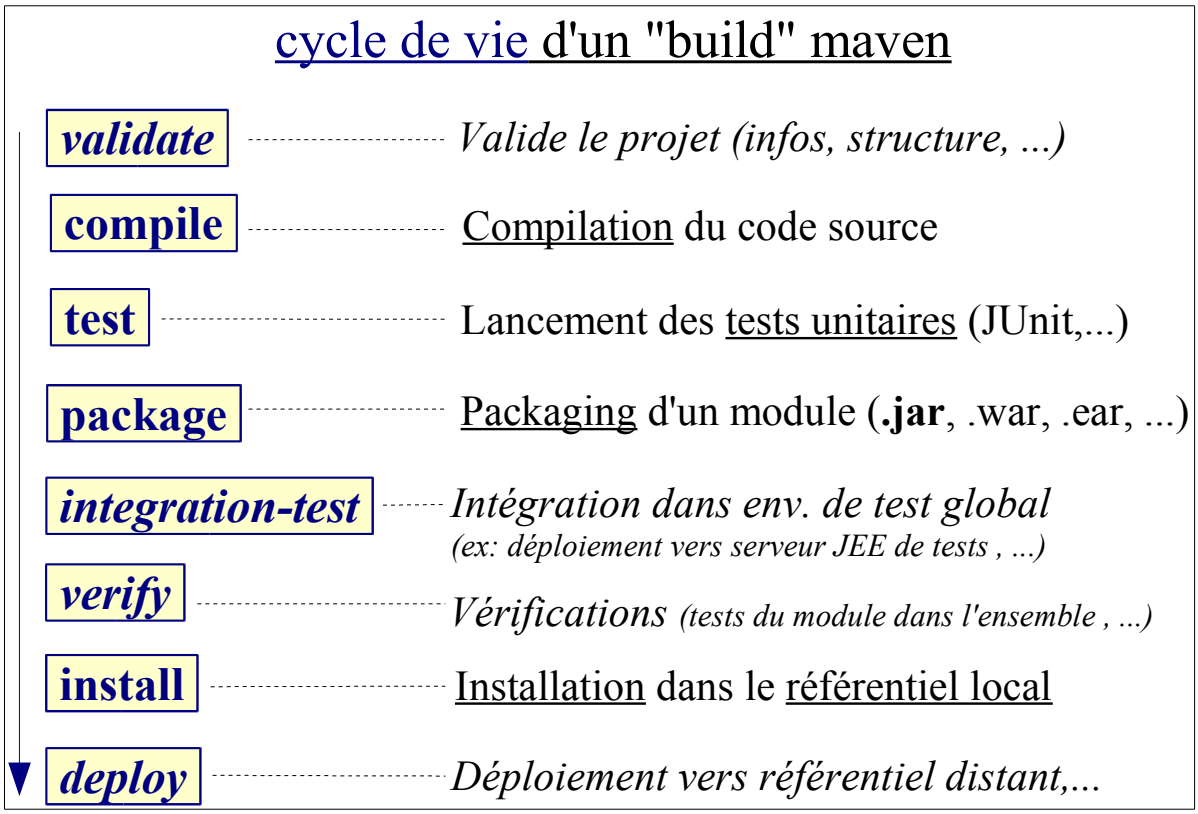
- Il existe un framework additionnel facultatif appelé "**dbUnit**" pour effectuer en plus des tests sur le contenu effectif d'une base de données .
  dbUnit est par exemple très pratique pour:
  * initialiser le contenu d'une base de données via des fichiers xml (en début de test)
  * vérifier qu'une action d'insertion ou de suppression a bien changé le contenu de la base de données (enregistrement en plus ou en moins)
  * ....

XII - Annexe – Strict essentiel MAVEN

1. Projet "maven" et dépendances

1.1. Présentation rapide de maven

Maven est une technologie très sophistiquée permettant de construire un projet java (compilation, tests , packaging, ...).
Ci dessous figure le strict essentiel pour une utilisation en mode développement sous eclipse.



En ligne de commande ou bien intégré dans eclipse

Maven est avant tout un utilitaire que l'on déclenche via des ligne de commandes
(ex: ***mvn package*** *ou* ***mvn clean***).

Il existe un plugin "**m2e**" permettant d'intégrer et gérer des projets "maven" au sein d'eclipse.

Sous eclipse :

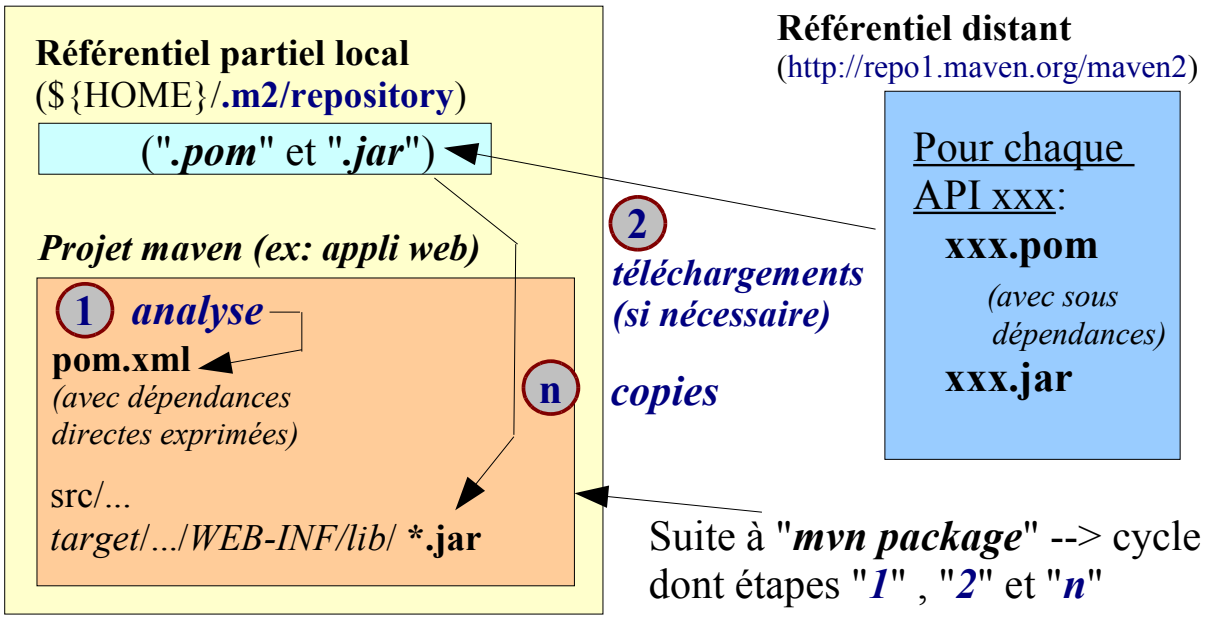
New projects.... /Maven project (souvent avec "skip archetype").

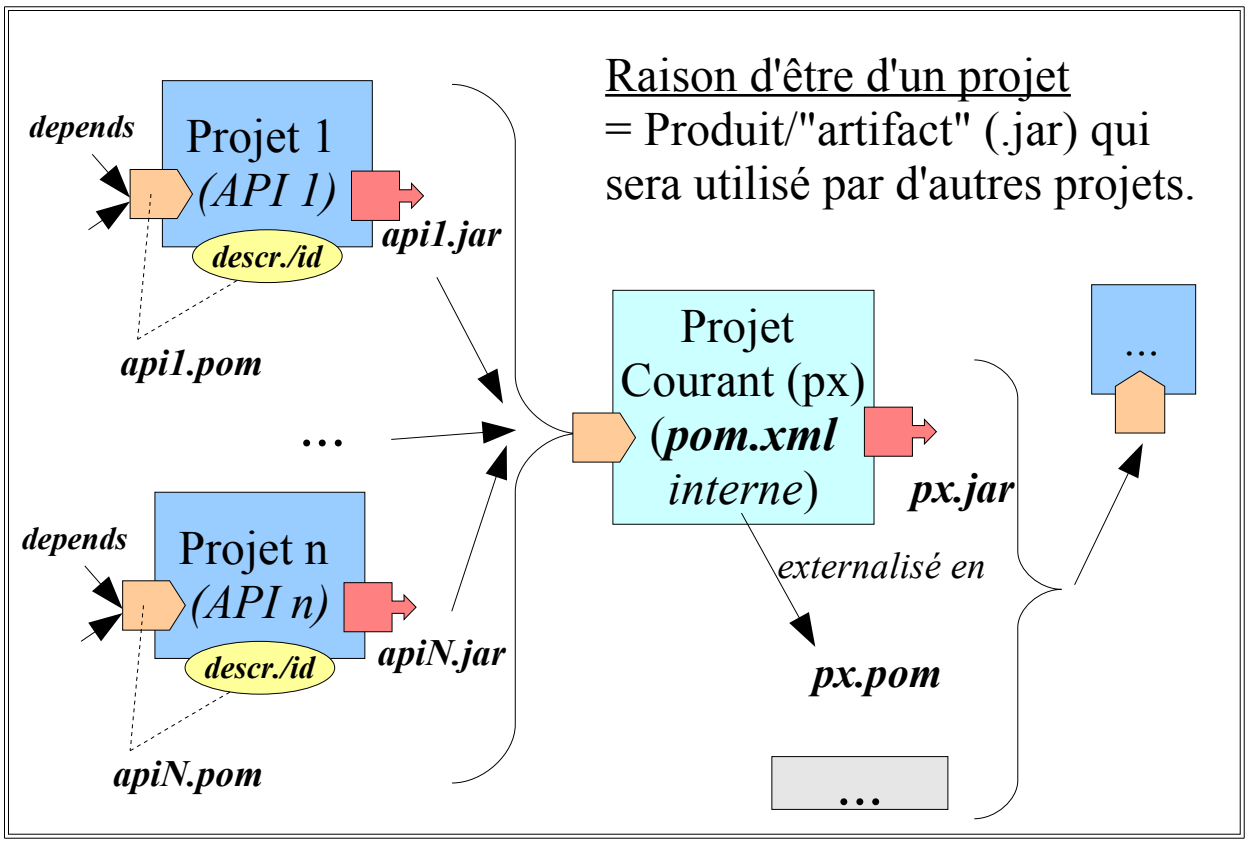
Si changement en profondeur du fichier "pom.xml" ===> **maven / update project configuration** .

Pour déclencher les constructions "maven" ===> **Run as / mvn package , ..**

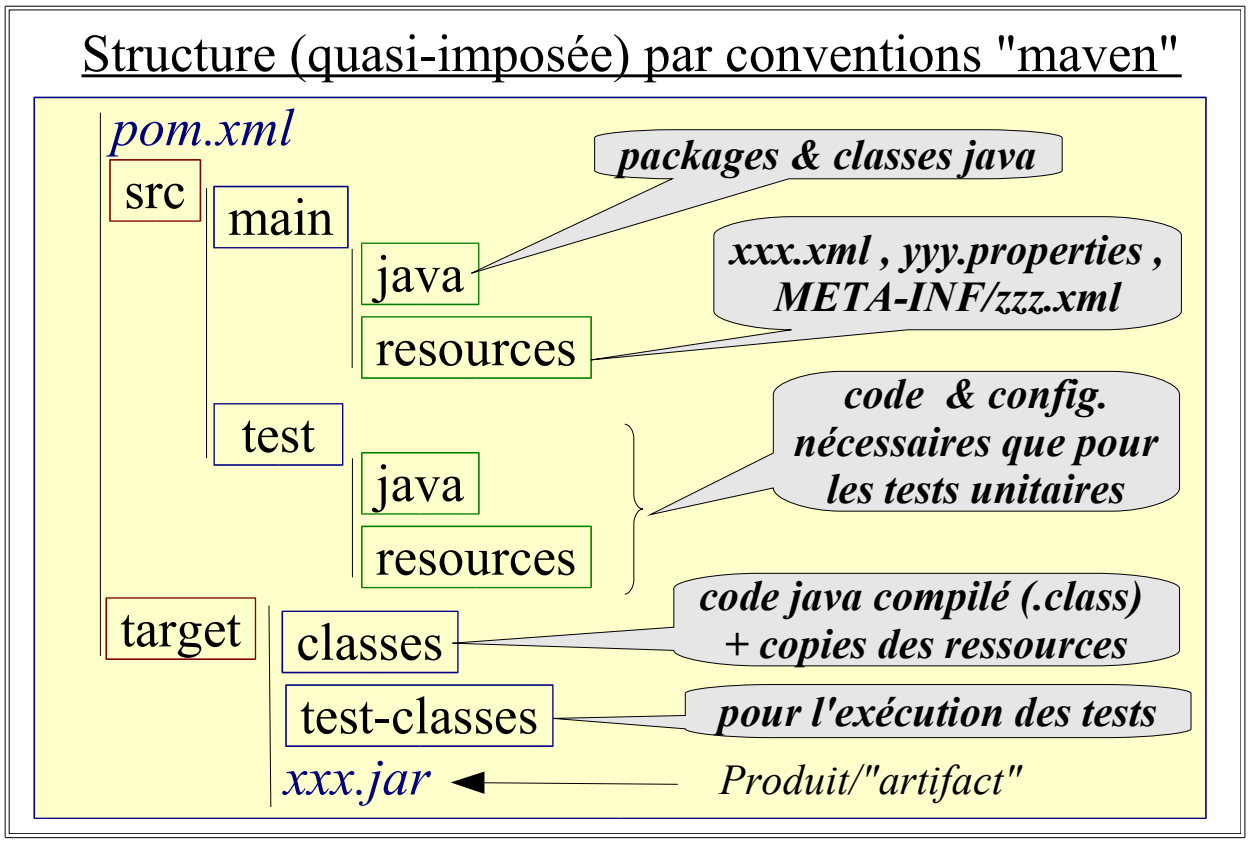
Téléchargement automatique des librairies nécessaires

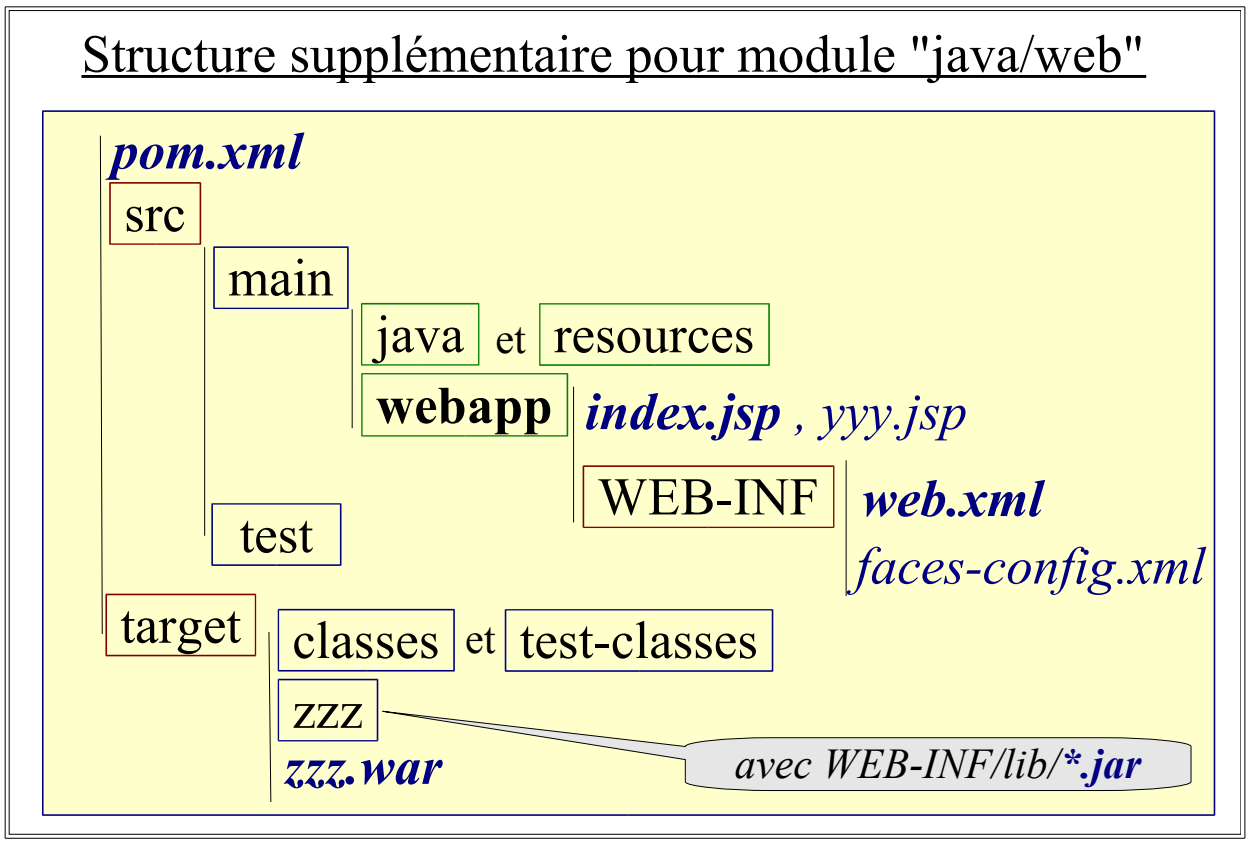
(avec prise en compte des dépendances indirectes)



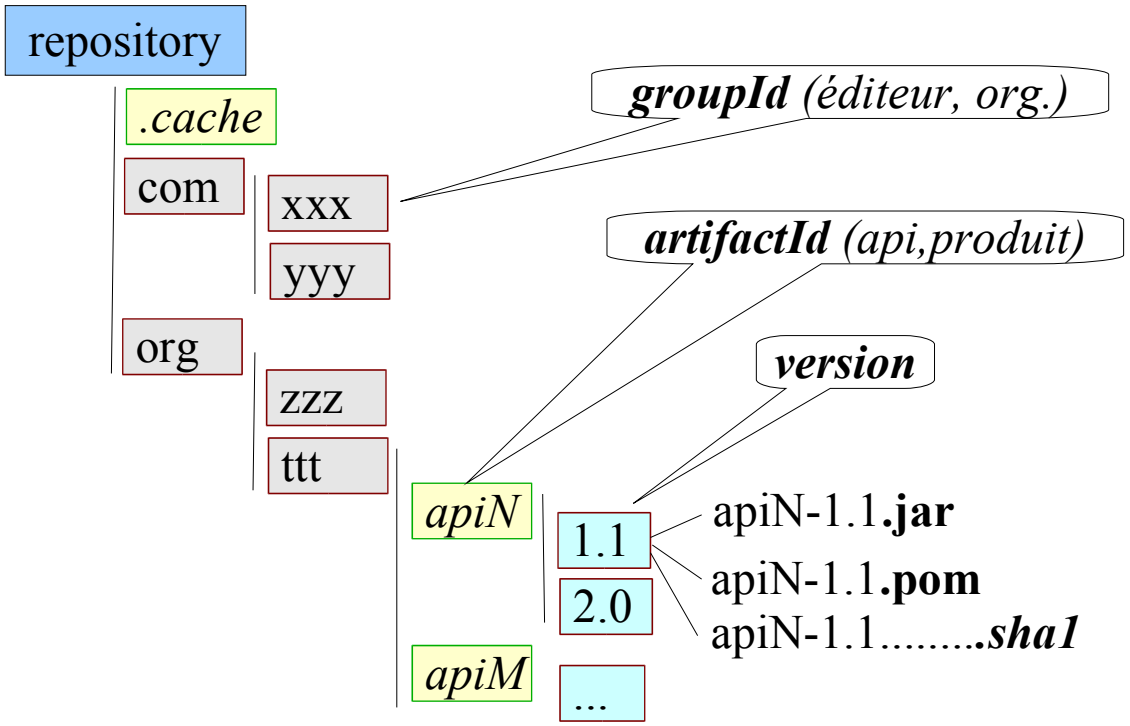


1.2. Arborescences conventionnelles





Structure d'un référentiel "maven" *(local ".m2" ou distant)*



1.3. portées ("scope") des dépendances:

Principaux types de dépendances "maven" (*scope*)

. **compile** *(par défaut)*
--> ***nécessaire pour l'exécution et la compilation*** *(dépendance directe puis transitive)* [diffusé dans tous les "classpath"].

. **runtime**
--> ***nécessaire à l'exécution*** *(dépendance indirecte* ***transitive****)*

.**provided**
--> ***nécessaire à la compilation*** mais ***fourni par l'environnement d'exécution (JVM + Serveur JEE)*** *[diffusé uniquement dans les "classpath" de compilation et de test, dépendance non transitive]*

. **test**
--> uniquement nécessaire pour les ***tests*** *(ex: spring-test.jar , junit4.jar)*

Diffusé dans quel(s) "classpath" ?

Type de dépendances	compilation	Tests unitaires	exécution	Transitivité (dans futur projet utilisateur / propagation)
compile (C)	x	x	x	C(C) -->C(*), P(C) -->P T(C) -->T, R(C) -->R
provided (P)	x	x	x *(provided)*	--> pas propagé , à ré-expliciter si besoin
test (T)	x	x		--> pas propagé
runtime (R)		x	x	R(R) --> R, C(R)-->R T(R)-->T, P(R)-->P

(*) bizarrement quelquefois "compile" plutôt que "runtime" dans le cas où l'on souhaite ultérieurement étendre une classe par héritage.

2. Structure & syntaxes (pom.xml)

Fichier "POM" (éléments essentiels)

servlet-api-2.3.pom (exemple)

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>javax.servlet</groupId>
  <artifactId>servlet-api</artifactId>
  <version>2.3</version>
</project>
```

Version du modèle interne de maven

biblio-web-....pom

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                         http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>  <parent> ... </parent>
  <groupId>tp</groupId>  <artifactId>biblio-web</artifactId>
  <packaging>war</packaging>  <version>0.0.1-SNAPSHOT</version>
  <name>biblio-web JEE5 Webapp</name>
  .... <dependencies> ... </dependencies>  <build>....</build>
</project>
```

Fichier "POM" (déclaration des dépendances / partie 1)

```
<project ...>
 <modelVersion>4.0.0</modelVersion> <parent> ... </parent>
 <groupId>tp</groupId>  <artifactId>biblio-web</artifactId>
 <packaging>war</packaging> <version>0.0.1-SNAPSHOT</version>
 <name>biblio-web JEE5 Webapp</name>....
<dependencies>
      <dependency>
                  <groupId>javax.servlet</groupId>
                  <artifactId>servlet-api</artifactId>
                  <version>2.5</version>      <scope>provided</scope>
       </dependency>
       <dependency>
                  <groupId>org.slf4j</groupId>
                  <artifactId>slf4j-api</artifactId>
                  <version>1.5.6</version>    <scope>compile</scope>
       </dependency> ...
</dependencies>
 <build>....</build>
</project>
```

Fichier "POM" (déclaration des dépendances / partie 2)

```
...
<dependency>
      <groupId>org.hibernate</groupId>  <artifactId>hibernate-core</artifactId>
      <version>3.5.1-Final</version> <scope>compile</scope>
      <exclusions>
            <exclusion>
                  <groupId>javax.transaction</groupId>
                  <artifactId>jta</artifactId>
            </exclusion>
            <exclusion>
                  <groupId>asm</groupId>  <artifactId>asm</artifactId>
            </exclusion>
      </exclusions>
</dependency>
<dependency>
      <groupId>javax.transaction</groupId>
      <artifactId>jta</artifactId>  <version>1.1</version>
</dependency>
...
```

exclusion(s) explicite(s) de dépendance(s) indirecte(s) transitive(s)

Contrôle direct de la version souhaitée pour éviter des conflits ou des doublons

Mise au point des dépendances (partie 3)

- La ***mise au point des dépendances*** peut éventuellement être délicate en fonction des différents points suivants:
  - * potentiel ***doublon*** (2 versions différentes d'une même librairie) à partir de plusieurs dépendances transitives indirectes.
  - * potentiel ***conflit*** de librairie à l'exécution (incompatibilité entre une librairie "A" en version "runtime" et une librairie complémentaire "B" en version "provided" imposée par le serveur JEE)
  - * autres mauvaises surprises de "murphy" .
- Eléments de solutions:
  - * ***étudier finement les compatibilités/incompatibilités*** et re-paramétrer les "***version***" et "***exclusion***"
  - * (tester , ré-essayer , re-tester) de façon itérative

Fichier "POM" (partie "build")

```
<project ...>
 <modelVersion>4.0.0</modelVersion> <parent> ... </parent>
 <groupId>tp</groupId>  <artifactId>biblio-web</artifactId>
 <packaging>war</packaging> <version>0.0.1-SNAPSHOT</version>
 <name>biblio-web JEE5 Webapp</name>....
<dependencies>
      <dependency>...</dependency> ...
</dependencies>
 <build>
   <plugins>  <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId><ersion>2.0.2</version>
            <configuration>
                <source>1.6</source> <target>1.6</target>
            </configuration>
      </plugin>...  </plugins>
  <finalName>biblio-web</finalName>
</build>
</project>
```

XIII - Annexe – Annotations Java

1. Annotations : Présentation et intérêts

Les **annotations** (disponibles depuis la version 1.5) sont des ***informations textuelles (ressemblant un peu à des commentaires)*** qui sont ***insérées au sein du code source*** (*juste au dessus d'une classe , d'une méthode ou d'un attribut*) .

Ces **annotations** sont quelquefois bien **utiles** car elles **pourront** (selon leurs portées) **être ultérieurement analysées par** :

- des outils tel que **APT (Annotation Processing Tool)** assimilables à des pré-processeurs et ***permettant de générer dynamiquement certains fichiers de configurations*** (xml , J2EE, ...) ***et d'éventuelles autres classes ou interfaces java***.
- l'**api de réflexion/introspection** de façon à ***paramétrer certains mécanismes génériques*** (persistance?, cryptage?, ...)
- ....

On parle quelquefois en terme de **méta-programmation** lorsque l'on à recours aux annotions dans le cycle de développement.

Une solution open source dénommée ***xdoclet*** (bien antérieure au jdk 1.5) permettait déjà d'obtenir des résultats comparables à l'époque des jdk 1.3 et 1.4 .

On peut peut donc voir les annotations du jdk 1.5 comme une nouvelle alternative (normalisée/standardisée, améliorée et officielle) vis à vis de l'ancien xdoclet (projet "open source" et précurseur).

Vue l'assez grande adoption de xdoclet sur de nombreux projets, la transition sera certainement assez longue (due à l'inertie induite par un existant devant rester cohérent).

Syntaxe des annotations:

NB: L'annotation porte toujours sur l'élément qui suit (sur la ligne suivante!).

```
@MonAnnotationSansParametre
public class Cxx { ...}

@AutreAnnotationSansParametre
public int méthodeX { ...}

@AnnotationAvecUneSeuleValeur("valeur_unique_parametre")
public class Cxx { ...}

@AnnotationAvecParamtres(param1="valeur_param1", param2=VALEUR2_CONTANTE)
public class Cxx { ...}
```

2. Annotations et méta-annotations prédéfinies

2.1. Annotations standards (interprétées par le compilateur):

Annotations standards du jdk 1.5	*Significations*
@Deprecated	Éléments (méthode , attribut, ...) devenu obsolète et qui ne devrait plus être utilisé. NB: l'utilisation conjointe du *commentaire javadoc @deprecated* permet en plus d'indiquer la raison et une suggestion de remplacement: ***/***** **** @deprecated pas bien car ... , utiliser .... à la place*** *****/*** ***@Deprecated***
@Override	Pour que le compilateur vérifie que la méthode qui suit est bien une redéfinition d'une méthode héritée existante (et pas une nouvelle opération) ==> ceci permet de détecter des erreurs sur la signature.
@SuppressWarnings	Pour demander au compilateur d'ignorer certains Warnings sur l'élément (classe,...) qui suit sans pour autant ignorer les les warnings sur le reste de l'application --> ex: ***@SuppressWarnings({"deprecation","unckeked"})***

2.2. Méta-annotations (Annotations sur annotations):

Nécessite ========> ***import java.lang.annotation.*;***

exemple:

@Documented

public interface @MyDocumentedAnnotation { ...}

Méta-Annotations du jdk 1.5	*Significations*
@Documented	L'annotation doit apparaître dans la documentation générée par javadoc ou ...
@Inherit	L'annotation sera automatiquement héritée par les sous classes (@Inherit ne peut être utilisée que sur annotation de classe – ce n'est pas utilisable sur une annotation d'interface).
@Retention(...)	Pour indiquer la portée (ou durée de vie) de l'annotation. **RetentionPolicy.SOURCE** (code source uniquement) **RetentionPolicy.CLASS** (dans .class également mais pas pris en compte par la JVM , c'est la valeur par défaut). **RetentionPolicy.RUNTIME** (vu par la JVM et l'api "reflection").

Méta-Annotations du jdk 1.5	*Significations*
@Target(...)	Pour que l'annotation ne soit utilisable que devant une ou plusieurs catégories d'éléments (ex: classe , méthode , ...). Si une annotation n'est pas bien utilisée (erreur de @Target) ==> Erreur du compilateur. Si la méta-annotation @Target n'est pas précisée , l'annotation peut être utilisée sur n'importe quel élément . **ElementType.ANNOTATION_TYPE** (devant annotation) **ElementType.CONSTRUCTOR** , **ElementType.FIELD** , **ElementType.LOCAL_VARIABLE** , **ElementType.METHOD** , **ElementType.PACKAGE** **ElementType.PARAMETER** **ElementType.TYPE** (devant classe , interface ou enum).

3. Création de nouvelles annotations

Une **nouvelle annotation** se code comme une **interface spéciale** ayant un **nom commençant par le caractère @** .

Une interface d'annotation est très souvent précédée par une ou plusieurs méta-annotations.

D'autre part, les éventuelles propriétés (implicitement "public") d'une annotation se codent comme des méthodes (implicitement abstraites) de l'interface d'annotation.

Le nom implicite d'une unique propriété est "***value***=" ==> ***value()*** ;

Exemples:

```
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.SOURCE;

@Documented
@Retention(SOURCE)
public interface @Important {
}
```

```
@Documented
@Retention(SOURCE)
public interface @TODO {
      /** Message décrivant la tâche à effectuer. */
      String value();
     /** Niveau de criticité de la tâche (défaut : NORMAL). */
      Level  level() default  Level.NORMAL;
      /** Enumération des différents niveaux de criticités. */
      public static enum  Level { MINEUR, NORMAL, IMPORTANT };
}
```

Attention ! Les attributs d'une annotation n'acceptent que les éléments suivants :

- Un type primitif (**boolean**, **int**, **float**, etc.).
- Une chaîne de caractères (**java.lang.String**).
- Une référence de classe (**java.lang.Class**).
- Une Annotation (**java.lang.annotation.Annotation**).
- Un type énuméré (**enum**).
- Un tableau à une dimension d'un des types ci-dessus.

Toutes les annotations héritent implicitement de l'interface **java.lang.annotation.Annotation**.

4. Insertion d'annotations au sein d'un code source

```
...
    @Important
     public class Cxx {
          /* ... */
     }
```

```
..
     @TODO(value="La gestion des exceptions est ...", level=NORMAL)
     public void methode1() {
          /* ... */
    }

    // étant donnée que Level.NORMAL est la valeur par défaut de la propriété level
    // on peut également écrire:

    @TODO(value="La gestion des exceptions est à améliorer ...")
     public void methode1() {
          /* ... */
     }

     // étant donnée que value est le nom (par défaut) d'une unique propriété, on peut également écrire:

     @TODO("La gestion des exceptions est à améliorer ...")
     public void methode1() {
          /* ... */
     }
```

5. Analyse et traitement des annotations via l'utilitaire APT (Annotation Processing Tool)

Le nouvel outil **APT** (**A**nnotation **P**rocessing **T**ool) du **JDK 5.0** permet d'effectuer des traitements sur les annotations avant la compilation effective des sources Java.
Il est ainsi capable de générer :

- des messages (note, warning et error).
- des fichiers (texte, binaire, source et classe Java).

Ceci avant de réellement compiler les fichiers *.java.

Pour cela, **APT** utilise les mêmes options de la ligne de commande que **javac**, avec en plus les suivantes :

- **-s dir** : Spécifie le répertoire de base où seront placés les fichiers générés (par défaut dans le même répertoire que **javac**).
- **-nocompile** : Ne pas effectuer la compilation après le traitement des annotations.
- **-print** : Affiche simplement une représentation des éléments annotés (sans aucun traitement ni compilation).
- **-A[key[=val]]** : Permet de passer des paramètres supplémentaires destinés aux *"fabriques"*.
- **-factorypath path** : Indique le(s) path(s) de recherche des *"fabriques"*. Si absent, c'est le classpath qui sera utilisé.
- **-factory classname** : Indique le nom de la classe Java qui servira de *"fabrique"*.

Toutefois si le paramètre **-factory** est absent, **APT** recherchera relativement dans les différents répertoires et archives jar du **classpath** *(ou du **factorypath** si précisé)* des fichiers nommés **com.sun.mirror.apt.AnnotationProcessorFactory** dans le répertoire **META-INF/services** .
Il s'agit d'un simple fichier texte contenant le nom complet des différentes *"fabriques"* qui seront utilisées. Il est ainsi possible d'en utiliser plusieurs .

Déclenchement de APT via un script ANT :

```
<javac fork="yes"    executable="apt"   srcdir="${src}"   destdir="${build}">
        <classpath>
            <pathelement path="xxx.jar"/>
        </classpath>
       <!- -  <compilerarg  value="-Arelease"/> - ->
</javac>
```

Utilisant les design patterns "*Factory*" et "*Visitor*" , **APT** a besoin des 3 éléments suivants:

- **AnnotationProcessorFactory** : La fabrique qui sera utilisée par **APT**.
- **AnnotationProcessor** créé par la fabrique et utilisé par **APT** pour traiter les fichiers sources.
- Les **Visitors** qui permettent de visiter simplement les différentes déclarations/types d'un fichier source.

Exemple de code pour un "***AnnotationProcessorFactory***"

```
public class SimpleAnnotationProcessorFactory implements AnnotationProcessorFactory {
	/** Collection contenant le nom des Annotations supportées. */
	protected Collection<String> supportedAnnotationTypes =
		Arrays.asList( TODO.class.getName(), Important.class.getName() );
	/** Collection des options supportées de APT ( -Akey[=value] ) */
	protected Collection<String> supportedOptions =
		Collections.emptyList();
	/**
	 * Retourne la liste des annotations supportées par cette Factory.
	 */
	public Collection<String> supportedAnnotationTypes() {
		return supportedAnnotationTypes;
	}
	/**
	 * Retourne la liste des options supportées par cette Factory.
	 */
	public Collection<String> supportedOptions() {
		return supportedOptions;
	}
	/**
	 * Retourne l'AnnotationProcessor associé avec cette Factory...
	 */
	public AnnotationProcessor getProcessorFor(Set<AnnotationTypeDeclaration> atds,
			AnnotationProcessorEnvironment env) {
		// Si aucune annotation n'est présente on retourne un processeur "vide"
		if (atds.isEmpty())
			return AnnotationProcessors.NO_OP;
		return new SimpleAnnotationProcessor(env);
	}
}
```

Exemple de code pour **AnnotationProcessor** :

```
public class SimpleAnnotationProcessor implements AnnotationProcessor {
	/** L'environnement du processeur d'annotation. */
	protected final AnnotationProcessorEnvironment   env;
	/**
	 * Constructeur.
	 * @param env L'environnement du processeur d'annotation.
	 */
	public SimpleAnnotationProcessor (AnnotationProcessorEnvironment env) {
		this.env = env;
	}


	/**  Traitement des fichiers sources balayés par un "Visiteur".   */
```

```
    public void process() {
        // Instanciation du Visitor
        TODOVisitor todoVisitor = new TODOVisitor(env);
        // On boucle sur tous les types d'annotations :
        for ( Declaration d : env.getTypeDeclarations()) {
            // On "visite" chacune des déclarations trouvées :
            d.accept( DeclarationVisitors.getSourceOrderDeclarationScanner(
                    todoVisitor, DeclarationVisitors.NO_OP) );
        }
    }
}
```

Exemple de code pour un **visiteur** (actif) *de déclarations (de classes , de méthodes , de ...)* :

```
public class TODOVisitor extends SimpleDeclarationVisitor{
    protected final AnnotationProcessorEnvironment env;
    public TODOVisitor (AnnotationProcessorEnvironment env) {
        this.env = env;
    }
    /**
     * Pour tout type de déclaration, on affiche un message si
     * l'Annotation @TODO est présente...
     */
    @Override
    public void visitDeclaration(Declaration decl) {
        // On regarde si la déclaration possède une annotation TODO
        TODO todo = decl.getAnnotation(TODO.class);
        // Et on l'affiche éventuellement :
        if (todo!=null)
            printMessage(decl, todo);
    }

    /** Affiche dans la console l'annotation TODO.   */
    public void printMessage (Declaration decl, TODO todo) {
        Messager m = env.getMessager();
        switch (todo.level())
            {
             case IMPORTANT:
                m.printWarning(decl.getPosition(), decl.getSimpleName() + " : " + todo.value() );
                break;
            case NORMAL:
            case MINEUR:
                m.printNotice(decl.getSimpleName() + " : " + todo.value() );
                break;
             }
    }
...
```

6. Accès aux annotations (de rétention RUNTIME) via l'introspection de java 5

L'API de Réflection (java.lang.reflect) a été entendue en version 1.5 de façon à supporter les annotations. Pour cela, les classes **Package**, **Class**, **Constructor**, **Method** et **Field** possèdent quatre nouvelles méthodes décrite dans l'interface **AnnotatedElement** :

- **getAnnotation(Class<A>)** qui retourne l'annotation dont le type est passé en paramètre (ou *null* si cette annotation n'est pas présente).
- **getAnnotations()** qui retourne un tableau comportant une instance de toutes les annotations de l'élément.
- **getDeclaredAnnotations()** qui retourne un tableau comportant une instance de toutes les annotations directement présentes sur l'élément (c'est à dire sans les annotations héritées de la classe parente). Rappel: Seules les **Class** peuvent hériter d'une annotation.
- **isAnnotationPresent(Class<A>)** qui retourne un booléen indiquant si l'annotation dont le type est passé en paramètre est présente sur l'élément ou non. Cette méthode est surtout utile pour les annotations marqueurs (sans attribut).

Exemple:

```
// Instanciation de l'objet:
Exemple objet = new Exemple();

// On récupère la classe de l'objet :
Class<Exemple> classInstance = objet.getClass();

// On regarde si la classe possède une annotation :
MonAnnotation annotation = classInstance.getAnnotation(MonAnnotation.class);

if (annotation!=null) {
      System.out.println ("MonAnnotation : " + annotation.value() );
      }
```

XIV - Annexe – Java Native Interface (c/c++)

1. JNI : Présentation , intérêts et dangers

Le langage JAVA donne la possibilité d'appeler; à partir de certains morceaux de code JAVA, des ***fonctions (méthodes) écrites en C ou C++.***

Ces méthodes sont dites natives et doivent être déclarées comme telles en utilisant le mot clé **native**.

Nb: Il ne faut surtout pas abuser des méthodes natives car elles ne sont pas indépendantes de la plate- forme (Unix ou Win32, …).

2. Déclaration et appel d'une méthode native

```
public class MaClasseJava
        {
        int x = 10;
        private native int aff() ;
        public static void main ( String [] argv)
                {
                // Chargement de la librairie dynamique
                // contenant le code compilé de la méthode native:
                System.loadLibrary("MaLibrairieNative");

                MaClasseJava instance = new MaClasseJava();
                instance.aff();
                }
        }
```

3. Implémentation d'une méthode native

La fonction doit être écrite en C. Il faut pour cela, connaître le prototype exact de la fonction. Ce prototype est particulier dans le sens ou il doit s'interfacer avec un objet Java dont l'ensemble des attributs doit être vu depuis le C sous la forme d'une structure.

On obtient ce prototype en utilisant l'outil **javah** suivi du nom de la classe qui va générer automatiquement un fichier d'entête (.h) contenant le prototype de la fonction C. Il faut avoir préalablement compilé le source Java avec l'outil **javac**.
Il ne reste alors plus qu'à écrire le corps de la fonction.

javah nompaquet.Nomclasse

Exemple de fichier header généré par javah (version JDK 1.2 / JNI) :

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class p1_MaClasseJava */

#ifndef _Included_p1_MaClasseJava
#define _Included_p1_MaClasseJava
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     p1_MaClasseJava
 * Method:    aff
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_p1_MaClasseJava_aff
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

code de la méthode native:

```
#include <stdio.h>
#include "p1_MaClasseJava.h"

JNIEXPORT jint JNICALL Java_p1_MaClasseJava_aff
  (JNIEnv * env, jobject obj)
{
jclass cls = env->GetObjectClass(obj);
jfieldID champX=env->GetFieldID(cls,"x" /*field name */,"I" /*signature*/);
jint x = env->GetIntField(obj,champX);
printf(" La valeur de l'attribut x est %ld \n", x);
return 0;
}
```

4. Etablir le lien entre JAVA et le code C:

Le code C doit être compilé sous la forme d'une librairie dynamique (.dll sous windows , so sous unix) qui sera chargée au début de l'exécution du programme Java.

Nb: Dans l'environnement Windows 32 bits, une DLL peut être fabriquée à partir du produit Visual C++ (projet de type DLL sans MFC) ou bien DevCpp.

XV - Annexe – package "java.net" (sockets, http)

1. Réseau / URL / Http

Remarque: tous les éléments de ce chapitre peuvent être aussi bien mis en oeuvre au sein:

- D'une application java autonome (main() en mode texte ou mode graphique).
- D'un applet java (en limitant les connexions vers le serveur source du téléchargement).
- D'un éventuel composant web (servlet) voulant établir une connexion avec un autre serveur web.

2. Récupération du contenu référencé par une URL.

```
import java.net.*;

String  adresse = "http://www.xxx.com/repX/fichier1.txt"

URL url = new URL(adresse);
String  contenuFichier = (String) url.getContent();
```

NB: La méthode ***getContent***() ne fonctionne correctement que si le contenu est d'un des types suivants:

- text/plain
- image/jpeg
- image/gif

3. Contrôle de la connexion liée à une URL

```
import java.net.*;

On peut récupérer des informations (type, taille, …) sur une URL au moyen d'une instance de la classe URLConnection.
…
URL url = new URL(adresse);
URLConnection conn = url.openConnection();
…
System.out.println(conn.getURL().toExternalForm() + ":");
System.out.println(" Content Type : " + conn.getContentType());
System.out.println(" Content Length : " conn.getContentLength());
System.out.println(" LastModified : " + new Date(conn.getLastModified()));
System.out.println(" Expiration : " + conn.getExpiration());
System.out.println(" Content Encoding : " + conn.getContentEncoding());

// Récupérer les 5 premières lignes du contenu au bout de l'URL:
DataInputStream in = new DataInputStream(conn.getInputStream());
for(int i=0; i<5;i++)
{ String line = in.readline();
  if(line == null) break;
  System.out.println(line);
```

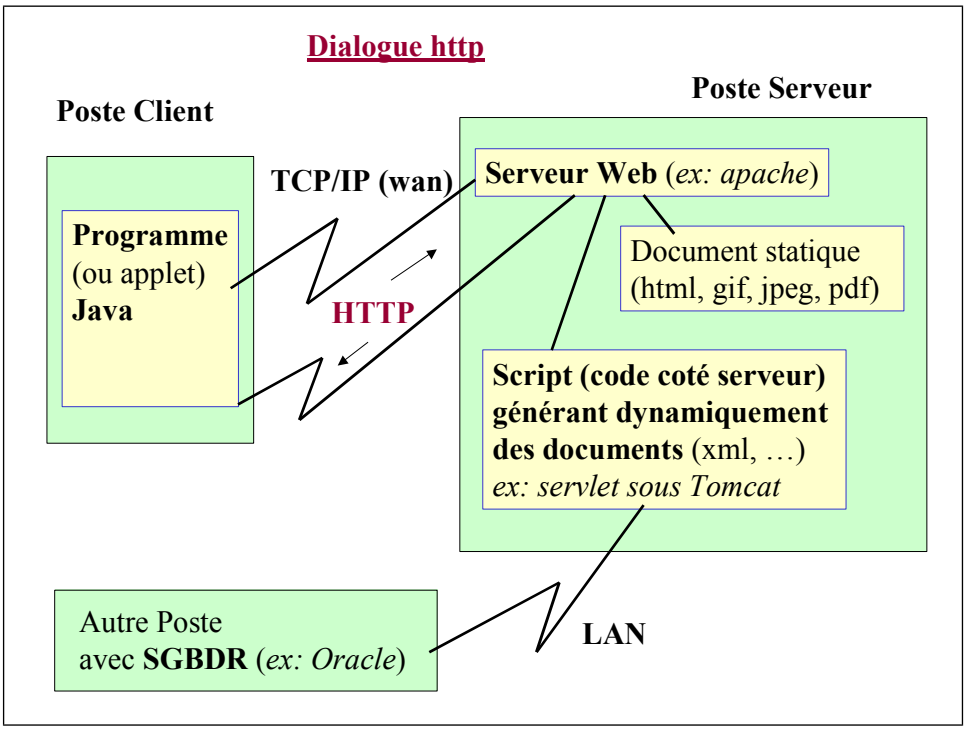
}

4. Présentation des quelques api "réseaux"

API	Caractéristiques
Socket (java.net)	Api de très bas niveau (juste au dessus de TCP/IP) . n° de port à spécifier. Api disponible depuis le jdk 1.0
RMI (java.rmi)	Appel de méthodes (à travers le réseau) sur des objets distants (qui s'exécutent dans une machine virtuelle java située sur une autre machine).
iiop (corba)	Api permettant de dialoguer avec des programmes écrits en C++ ou dans un langage quelconque (voir CORBA del'OMG).
Service web / **JAX-WS**	Appel de procédures distantes via SOAP (utilisation conjointe de HTTP et d'XML pour le format des requêtes et des réponses)

5. Dialogue HTTP entre un applet JAVA et un servlet

Un applet Java (ou bien une application autonome) peut très bien envoyer une **requête http de type POST** vers un servlet ou à un équivalent (script CGI, page ASP, page JSP, page Php, ...). Celui-ci va alors en retour renvoyer une réponse http vers notre applet ou application JAVA.



Bien qu' **HTTP** soit un protocole sans connexion, il offre néanmoins les avantages suivants:

- très simple.
- universellement reconnu (vrai standard)
- protocole accepté par les "firewall"

Il est également important de souligner **que le protocole HTTP ne fait que spécifier le format des entêtes des requêtes et des réponses**. **Les formats des contenus (corps) peuvent être quelconques** (text/plain ,**text/xml**, text/html , **application/octet-stream** , ...).

L'exemple suivant montre le code type (coté client) permettant un dialogue http / servlet :

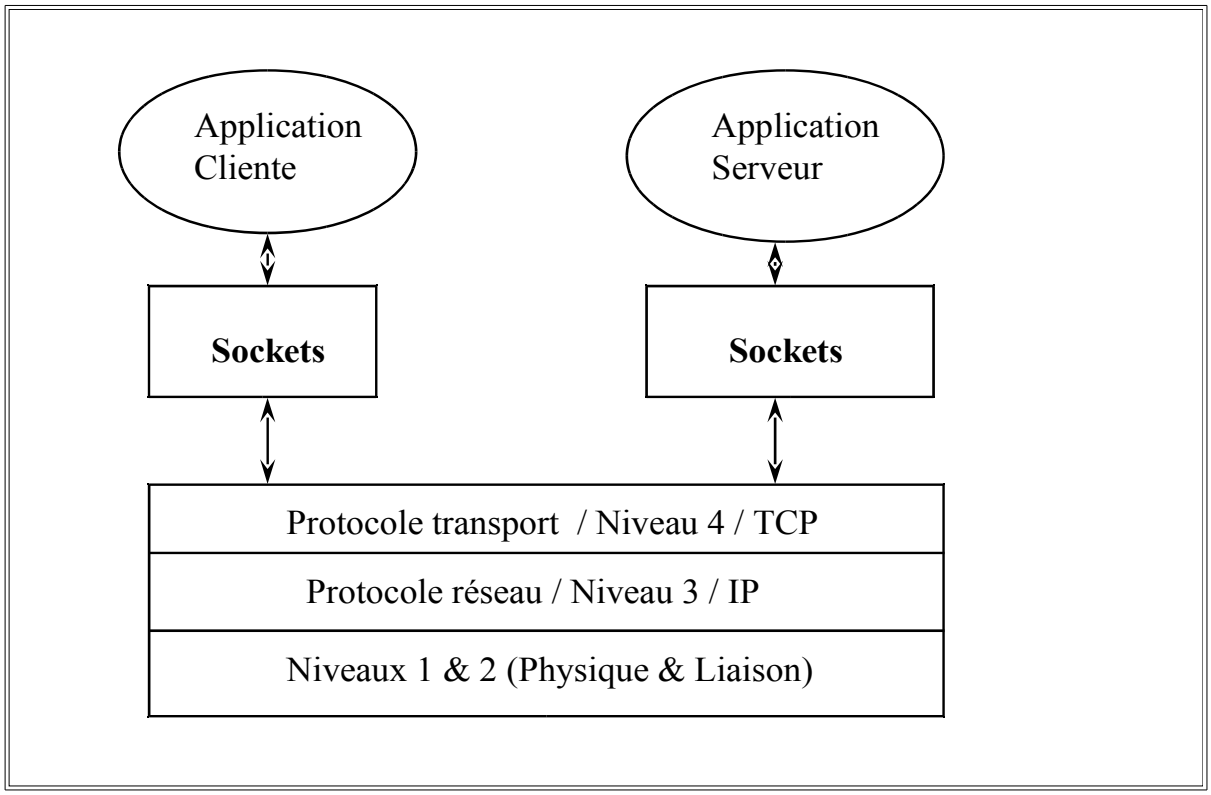
```
import java.net.*;
...
URLConnection urlConnect; // connection Http
int tailleReq; // taille de la requete
...
  // Paramétrage de la connection HTTP:
   try {
   //String chContentType="text/plain";
   String chUrl = " http://www.xxx.fr/siteY/servlet/nomLogiqueDuServlet";
   URL url =new URL(chUrl);

   urlConnect = url.openConnection();
   //urlConnect.setRequestProperty("Content-Type" ,chContentType);
   urlConnect.setDoOutput(true);
   //urlConnect.setDoInput(true); // true par défaut (pour récupérer la réponse)

   // Connection HTTP:
   PrintStream fluxHttp1;
   try {
      fluxHttp1 = new PrintStream(urlConnect.getOutputStream());
      // NB: l'appel à getOutputStream génère une connexion en mode POST
     } catch(Exception except) {."La connexion a échoué : " + except.toString()... }

   // Envoi des données au servlet :
   fluxHttp1.println("param1=valeur1&param2=valeur2");
   fluxHttp1.close();

   // Récupérer le résultat (entête):
   //chContentType = urlConnect.getContentType();
   //int tailleRes=urlConnect.getContentLength(); // si info retournée par le serveur
   //if(tailleRes > 0)
   //{
   // Récupération du résultat (corps) :
   InputStream fluxHttp = urlConnect.getInputStream();
   BufferedReader fluxNews = new BufferedReader(new InputStreamReader(fluxHttp));
   while(true)
      {
      chLigne = fluxNews.readLine();
      if(chLigne == null) break;
      ...
      }
   fluxNews.close();
   fluxHttp.close();
   //}
...
```

NB: C'est ce genre de code (de bas niveau) qui est en interne utilisé au sein des services Web

6. Sockets JAVA (depuis JDK 1.0)

6.1. Communications réseaux via les sockets

Le terme socket désigne une API réseau d'assez bas niveau (juste au dessus de TCP/IP). Cette API a été développée dans le monde UNIX et a été reprise par Microsoft (WinSock). Presque toutes les communications TCP/IP utilisent directement ou indirectement les sockets.
Le package ***java.net*** comporte quelques classes permettant de gérer simplement les sockets (Socket,ServerSocket,DatagramSocket, …).



6.2. Protocoles / Utilisation des sockets

6.2.a. Notion d'adresse internet :

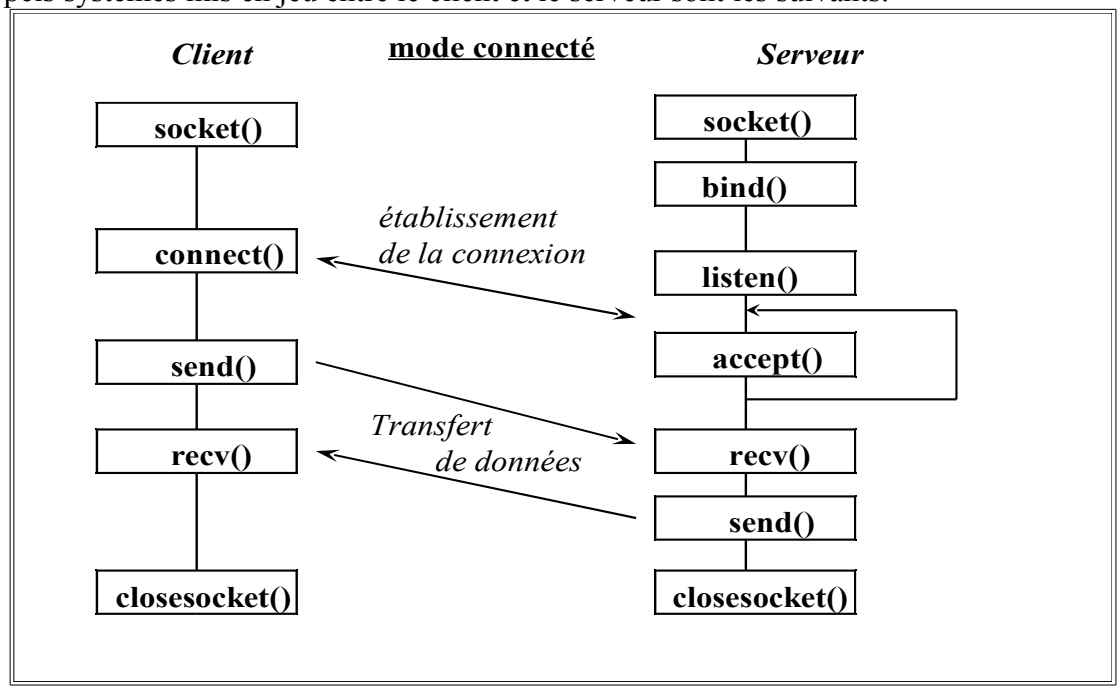
Pour identifier le service distant avec lequel il faut entrer en communication , l' API des sockets utilise la notion d' adresse internet. Il s'agit d'une structure de donnée qui regroupe les informations suivantes:

- Adresse IP de l'ordinateur (exemple: 128.127.12.10)
- N° de port du service (exemple: 9999)
- Protocole utilisé (UDP, TCP, …)

NB: Certains services ont des n° de port réservé (ex: http = 80 , ftp = 21 , telnet = 23 , …). Il faut donc utiliser des n° de port élevé pour ne pas entrer en conflit avec ceux qui sont prédéfinis (voir fichier [WinNt/system32/drivers]/etc/services).

6.2.b. Mode Connecté:

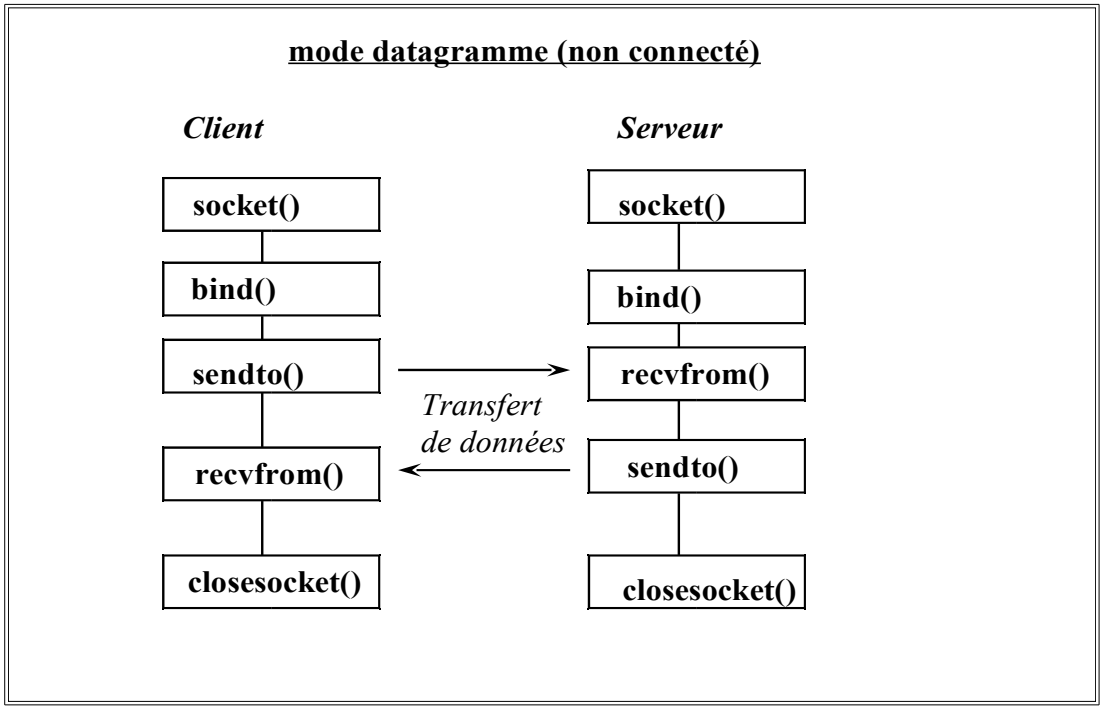
Ce mode d'utilisation des sockets s'appuie sur le protocole fiable TCP.
Les appels systèmes mis en jeu entre le client et le serveur sont les suivants:



NB: accept() duplique la socket d'origine du coté serveur. La socket d'origine sert alors à se mettre à l'écoute d'un autre client tandis que son homologue dupliquée sert à traiter la ou les requête(s) du client qui s'est connecté.

6.2.c. Mode non Connecté (DataGramme):

Le mode non connecté ne fait pas de distinction entre le client et le serveur. Il permet simplement d'envoyer un paquet d'information (appelé Datagramme) vers un autre processus distant.



6.3. Envoi d'un datagramme en mode non connecté:

```
int mumPortDest = 4567;  // identifie le service distant
String hostName = "MachineXXX";  // destinataire
String message = "Vive Java !!!";  // message à envoyer

int msglen = message.length();
byte [] buffer = new byte[msglen];
message.getBytes(0,msglen,buffer,0);  // conversion String → byte[]

// Initialise un objet "adresse IP" en fonction d'un nom de machine:
InetAddress address = InetAddress.getByName(hostName);

// Création du paquet à envoyer , on précise içi (sous JAVA) le n° de port:
DatagramPacket paquet =
        new DatagramPacket (buffer,msglen,address,numPortDest);

// Création d'une nouvelle socket en mode non connecté:
DatagramSocket socket = new DatagramSocket();

// Envoi du paquet:
socket.send(paquet);

….
```

6.4. Recevoir des datagrammes:

```
int monNumPort = 4567;       // numéro de port du receveur
byte[] bufferRecp = new byte[1024 /* ou plus */ ];  // buffer pour recevoir message

// création d'une socket associée au numéro de port du service que l'on gère:
DatagramSocket socket = new DatagramSocket(monNumPort);

for(;;)
{
DatagramPacket paquet = new DatagramPacket(buffer,buffer.length);
socket.receive(paquet); // reception (remplissage du paquet)

// conversion byte[]  → String
String s=new String(buffer,0,0,packet.getLength());

// Affichage de l'émetteur et du message reçu :
System.out.println("Received from " +
                        packet.getAddress().getHostName() + ":" +
                        packet.getPort() + ": " + s);
}
```

6.5. Serveur en mode connecté:

```
package serveur;

import java.lang.Thread;
import java.io.*;
import java.net.*;


class SockSrv extends Thread
{
public final static int DEFAULT_PORT = 7788;
protected int port;
protected ServerSocket listen_socket;

public static void main(String[] args)
        {
                int num_port = DEFAULT_PORT;
                if(args.length == 1)
                    try { num_port = Integer.parseInt(args[0]);}
                   catch(NumberFormatException e) {}
                   new SockSrv(num_port);
        }

public SockSrv(int num_port)
     {
          this.port = num_port;
          try { listen_socket = new ServerSocket(port); }
          catch(IOException e)
          { System.err.println("erreur création socket");
       System.exit(1); }
            System.out.println("Serveur en écoute sur le port " + port);
       this.start();
       }

public void run()
  {
          try { while(true)
         {
                Socket client_socket = listen_socket.accept();
                System.out.println("Un client s'est connecté");
                MySocketConnection c = new MySocketConnection(client_socket);
         }
       }
    catch(IOException e){
         System.err.println("erreur durant l'écoute"); System.exit(2); }
      }
}
```

```
package serveur;
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.Socket;

class MySocketConnection extends Thread
{
protected Socket client;
protected BufferedReader in;
protected PrintStream out;

// Constructeur qui initialise les flots d'entrée et sortie
// et qui s'auto-active (Thread).
public MySocketConnection(Socket client_socket)
            {
            client = client_socket;
            try { in = new BufferedReader(new InputStreamReader(client.getInputStream()));
                  out = new PrintStream(client.getOutputStream());}
            catch(IOException e)
                {try { client.close() ; } catch(IOException e2) {}
                 System.err.println("Erreur init flots "); return;
                }
             this.start();
            }

public void run()
          {
          String ligne, reponse;
          try {
               for(;;)
                    { ligne = in.readLine();
                      if(ligne == null) break; // déconnection ==> EOF ==> readline() retournant null
                      if (ligne.equals("Date")) reponse = "Aujourd'hui";
                         else if (ligne.equals("Heure")) reponse = "Maintenant" ;
                                 else reponse = "Réponse par défaut";
                      System.out.println("requete= "+ligne+" --> réponse= "+reponse);
                      out.println(reponse);
                    }
               }
          catch(IOException e) {}
          finally{
                   System.out.println("Un client s'est déconnecté");
                  try { client.close() ; } catch(IOException ec) {}
                 }
        }
}
```

6.6. Client (içi Application Swing) en mode connecté:

```
package client;

...
import java.net.Socket;

public class SockCli  extends  JFrame
{
public static final int PORT=7788;
Socket s;
BufferedReader in;
PrintStream out;
JTextField input_field , status_field;
JTextArea output_area;
MyStreamListener listener;
String defaultHost="localhost";    String serverHost = defaultHost;

public static void main(String args[])
{
        SockCli cliFrame = new SockCli();   cliFrame.init();
}

public void init()
{
        initGUI();          initSockets();
}

public void initGUI()
{

         input_field = new JTextField();    output_area = new JTextArea();
         status_field = new JTextField();
         output_area.setEditable(false);status_field.setEditable(false);
         output_area.setPreferredSize(new Dimension(250,80));
         this.getContentPane().setLayout(new BorderLayout());
         this.getContentPane().add("North",input_field);
         this.getContentPane().add("Center",output_area);
         this.getContentPane().add("South",status_field);
         this.setTitle("Client-sockets");     this.pack();       this.setVisible(true);
         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

         input_field.addKeyListener(new KeyAdapter()
                        {
                public void keyPressed(KeyEvent e)
                {
                  input_field_keyPressed(e);
                }
                        });
}

public void initSockets()
{
        try {
```

```
            s = new Socket(serverHost,PORT);
            in = new BufferedReader(new InputStreamReader(s.getInputStream()));
            out = new PrintStream(s.getOutputStream());
            listener = new MyStreamListener(in,output_area);

            status_field.setText("Connected to "
                        + s.getInetAddress().getHostName()
                        + ":" + s.getPort());
            }
            catch(IOException e) { status_field.setText(e.getMessage()); }
           }

private void input_field_keyPressed(KeyEvent e)
{
   if(e.getKeyCode()==KeyEvent.VK_ENTER) {
          out.println(input_field.getText());
          input_field.setText("");
   }
}
}
```

```
package client;

import java.io.BufferedReader;    import java.io.IOException;
import javax.swing.JTextArea;
```

```
public class MyStreamListener  extends Thread
          {
          BufferedReader in;        JTextArea output_area;

          public MyStreamListener(BufferedReader i,JTextArea t)
                    { in=i; output_area = t; this.start(); }

          public void run()
                    {
                    String ligne;
                    try {
                              for(;;)
                              { ligne=in.readLine(); if (ligne==null) break;
                              output_area.append(ligne+"\n"); }

                              } catch(IOException e)
                                        { output_area.setText(e.toString()); }
                    finally { output_area.setText("Connection closed by server."); }
                    }
          }
```

XVI - Annexe – Securité "java2" (.policy)

1. Sécurité Java2 : Présentation et utilité

Le jdk 1.0 a introduit la notion de bac à sable (sandbox) pour les applets: Un applet ne peut pas lire ou écrire sur le disque dur . ...

Le **jdk 1.1** a introduit la possibilité de **signer électroniquement le code téléchargé d'un applet** (archives CAB,ZIP ou JAR avec certificats) => ceci permet d'autoriser certains applets à outrepasser les restrictions classiques .

Le **jdk 1.2 (début de Java 2)** a enfin introduit un nouveau modèle de sécurité au sein duquel on peut **paramétrer finement les restrictions et/ou permissions d'accès que l'on accorde à tel morceau de code Java** (Application, Applet, Java Web Start , ...) .

2. Sécurité Java2 : Configuration & paramétrages

Sur le poste client final (où est lancé le navigateur Internet) , un fichier de nom [.]**java.policy** permet de **préciser** quelles seront les **permissions d'un applet** [ou d'une application "java_web_start"] (télé)chargé à partir de **tel emplacement** (et éventuellement accompagné d'une telle signature digitale).

NB : La signature électronique (avec certificat) n'est pas obligatoire. Elle permet néanmoins de rassurer l'utilisateur final sur les points suivants:
On est (relativement) sûr de l'origine de l'applet (édité par telle société).
Le code(ou exécutable) téléchargé ne peut pas avoir été facilement intercepté et modifié.

Vocabulaire:
CODEBASE =endroit depuis lequel on (télé)charge le code java.

La machine virtuelle Java 2 (Jre>=1.2) va rechercher les fichiers de configuration de la sécurité "Java 2" dans les répertoires suivants:

${JAVA_HOME}\lib\security\java.policy
(globalement valable pour tous les utilisateurs)
et **${HOME}\.java.policy**
(paramétrages spécifiques à un utilisateur)

*L'utilitaire **policytool.exe** du JDK permet de graphiquement configurer ce fichier .*

Précision sur la ligne de commande (lancement du programme):

```
java –Djava.security.policy=xxx.policy   AppliJava
```

*où **xxx.policy** peut être écrit de la façon suivante:*

```
grant {
   permission java.net.SocketPermission "*", "connect, resolve";
   permission java.io.FilePermission "<<ALL FILES>>", "read";
};
```

XVII - Annexe – Structure globale appli.

1. Architecture généralement recommandée

Partie "Présentation" ===> Partie "Traitements métiers" ===> Partie "Accès aux données"

Appli mode texte		**Calculs**		**Fichiers (txt,xml)**
ou		***et/ou***		***et/ou***
AWT/SWING	**===>**	**Assemblage/filtrage**	**===>**	**Base de données**
ou		***et/ou***		***et/ou***
....		**Règles de gestion , Mise à jour**		**Requêtes ...**

Exemples d'organisation pour le code:

- Plusieurs ***packages*** (bien séparés) : (*ex*: presentation , business , data)
- Classes avec responsabilités bien séparées (MyApp , MainFrame , XXXData , ...)
- Design Pattern "Singleton" pour accéder aux "Fabriques" d'accès aux données , ...
  ou bien Framework "IOC" avec conteneur léger (ex: Spring).
- Pas de valeur en dur mais fichiers de configuration "xxx.properties"ou "xxx.xml"
- ...

XVIII - Annexe – Prise en main de l'IDE Eclipse

1. Configurations [eclipse , projets]

1.1. Configurations générales / globales

Menu principal pour configurer globalement l'ensemble de l'environnement de développement :

==> **Window / Preferences**

Paramétrage clef **: sous menu "Java / install JRE " ==> chemin d'accès au JDK(s)**

Menu à retenir pour faire apparître une vue manquante dans la perspective courante**:**

==> Window / Show View

1.2. Création d'un nouveau projet

==> **File / New project / Java (or ...) project / ...**

NB: préférer séparer le répertoire source (**src**) du répertoire se sortie (code compilé / **bin**)

1.3. Compilation et lancement d'un programme:

Pour recompiler le projet courant:

==> **Project / Build** (ou simplement enregistrer le fichier source si ***project / build automatically***)

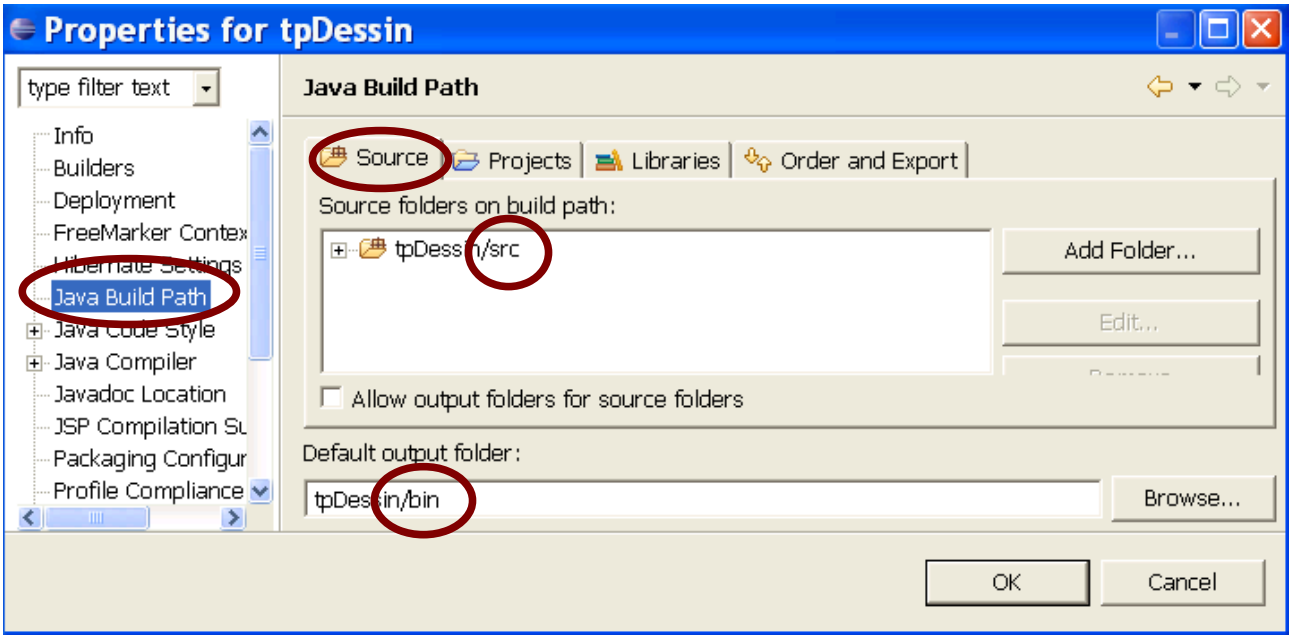
==> De temps en temps "***Refresh***" + **Project / clean** est utile pour forcer à recompiler tout le projet (lorsque le mode automatique optimise trop)

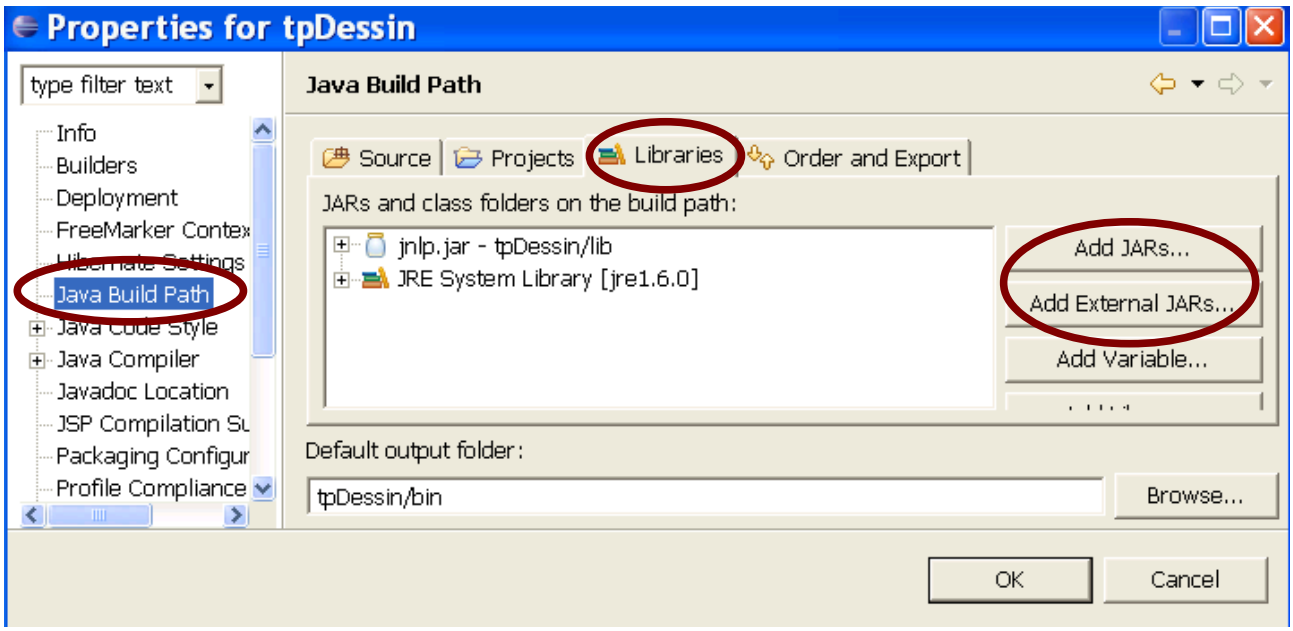
Pour lancer le programme courant:

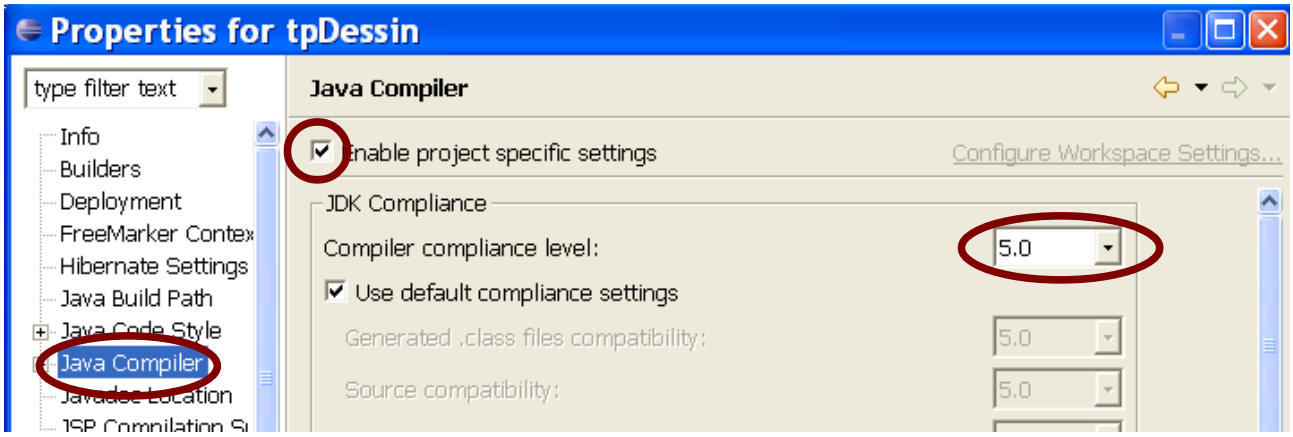
1) sélectionner la classe avec la méthode main(...)
2) **click droit / Run as Java Application** [+ paramétrages fin dans le menu Run/Run ...]

1.4. Configurations spécifiques à un projet

1) Sélectionner un projet
2) **click droit / Properties**


Properties for tpDessin
type filter text
Info
Builders
Deployment
FreeMarker Contex
Java Build Path
Java Code Style
Java Compiler
Javadoc Location
JSP Compilation Su
Packaging Configur
Profile Compliance
Java Build Path
Source
Projects
Libraries
Order and Export
Source folders on build path:
tpDessin/src
Add Folder...
Edit...
Allow output folders for source folders
Default output folder:
tpDessin/bin
Browse...
OK
Cancel


Properties for tpDessin
type filter text
Info
Builders
Deployment
FreeMarker Contex
Java Build Path
Java Code Style
Java Compiler
Javadoc Location
JSP Compilation Su
Packaging Configur
Profile Compliance
Java Build Path
Source
Projects
Libraries
Order and Export
JARs and class folders on the build path:
jnlp.jar - tpDessin/lib
JRE System Library [jre1.6.0]
Add JARs...
Add External JARs...
Add Variable...
Default output folder:
tpDessin/bin
Browse...
OK
Cancel


Properties for tpDessin
type filter text
Info
Builders
Deployment
FreeMarker Contex
Hibernate Settings
Java Build Path
Java Code Style
Java Compiler
Javadoc Location
JSP Compilation Su
Java Compiler
Enable project specific settings
Configure Workspace Settings...
JDK Compliance
Compiler compliance level:
5.0
Use default compliance settings
Generated .class files compatibility:
5.0
Source compatibility:
5.0

1.5. Edition du code source

1) **se placer dans un éditeur de code java**
2) **click droit / Source / ....**

Generate Getter/Setter ==> génère les méthodes public getXxx() et setXxx(...) en fonction des attributs privés "xxx"

Organize imports (Ctrl-Shift-O) ==> ajoute les "import ....;" manquants et retire les "import .....;" inutiles

....

1.6. Restructuration du code / refactoring

1. **Sélectionner un package ou une classe ou un membre (attribut/méthode)**
2. **click droit *Refactor/Rename ...* ==> renomme non seulement la chose sélectionnée mais aussi tous les autres morceaux de code du projet qui y font référence .**

2. IDE et Perspectives

L'environnement **eclipse** a la particularité d'être basée sur la notion de perspectives.

Une **perspective** est une **vision particulière du projet (sous un certain angle)**:

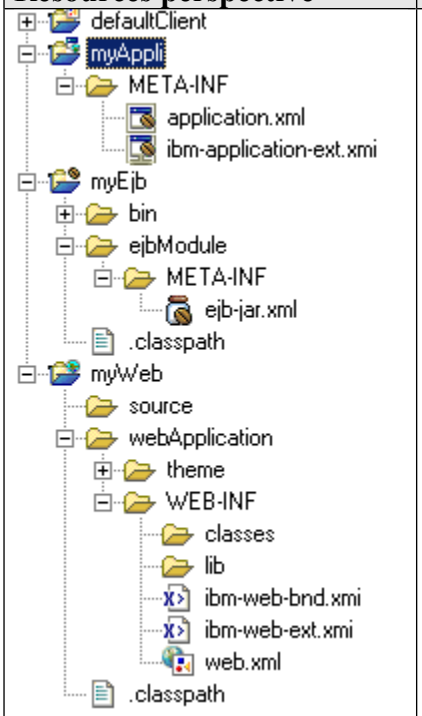
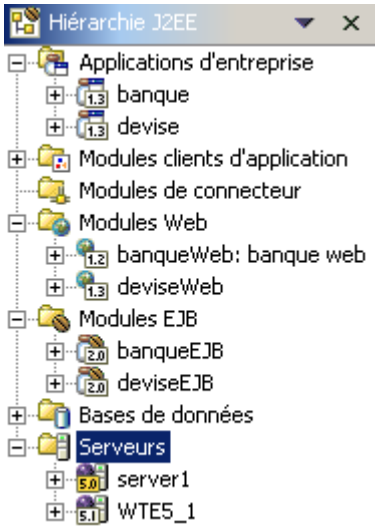
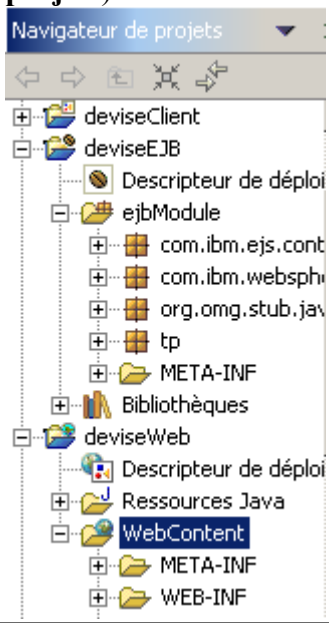
- Resources = Visualisation et navigation par fichiers et répertoire
- Java = Visualisation par packages et classes java
- JEE = Visualisation par modules et par composants "métier" (ejb, ...).
- Help = Visualisation des fichiers d'aide.
- Debug = Environnement pour effectuer les tests (avec console, ...).
- ...

On bascule d'une perspective à l'autre via des minis onglets situés par défaut sur le coté gauche.

Chacune de ces **perspectives** correspond à un **(sous) environnement de travail** et est composée d'un ensemble de *fenêtres* (appelées *Vues*).

La principale vue d'une perspective est généralement une **arborescence** permettant de naviguer à l'intérieur d'un ensemble d'entités pour en sélectionner une et ainsi afficher ses détails dans différentes vues secondaires.

==> L'environnement de développement comporte donc *n niveaux*:
perspectives
└ ***aborescences***
└ ***entités***
└ ***vues*** *(propriétés, code, ...)*
└ ***(éventuels onglets de la vue)***

Resources perspective	**JEE perspective** (*sous onglet* **Hiérarchie JEE**)	**JEE perspective** (*sous onglet* **Navigateur de projets**)
		

vision sous forme de **répertoires** et de **fichiers**

vision sous forme de **modules JEE** et de **serveurs de test** (ayant leur propre ***configuration*** [sécurité , pool de connexions , files JMS , ...])

vision sous forme de **projets JAVA** (packages , *classes* , fichiers de configurations , ...)

Remarque importante :

La perspective « **Ressources** » correspond à la vision totale du contenu physique du projet (**tous les répertoires**, **tous les fichiers** (.txt, .java, .html , …) , …).
Pour y introduire un nouveau fichier , on peut soit directement effectuer un **glisser/poser depuis l'explorateur de fichiers** , soit rafraîchir la liste des fichiers d'une partie du projet via le menu contextuel « **Refresh from Local** » de l'arbre des ressources.

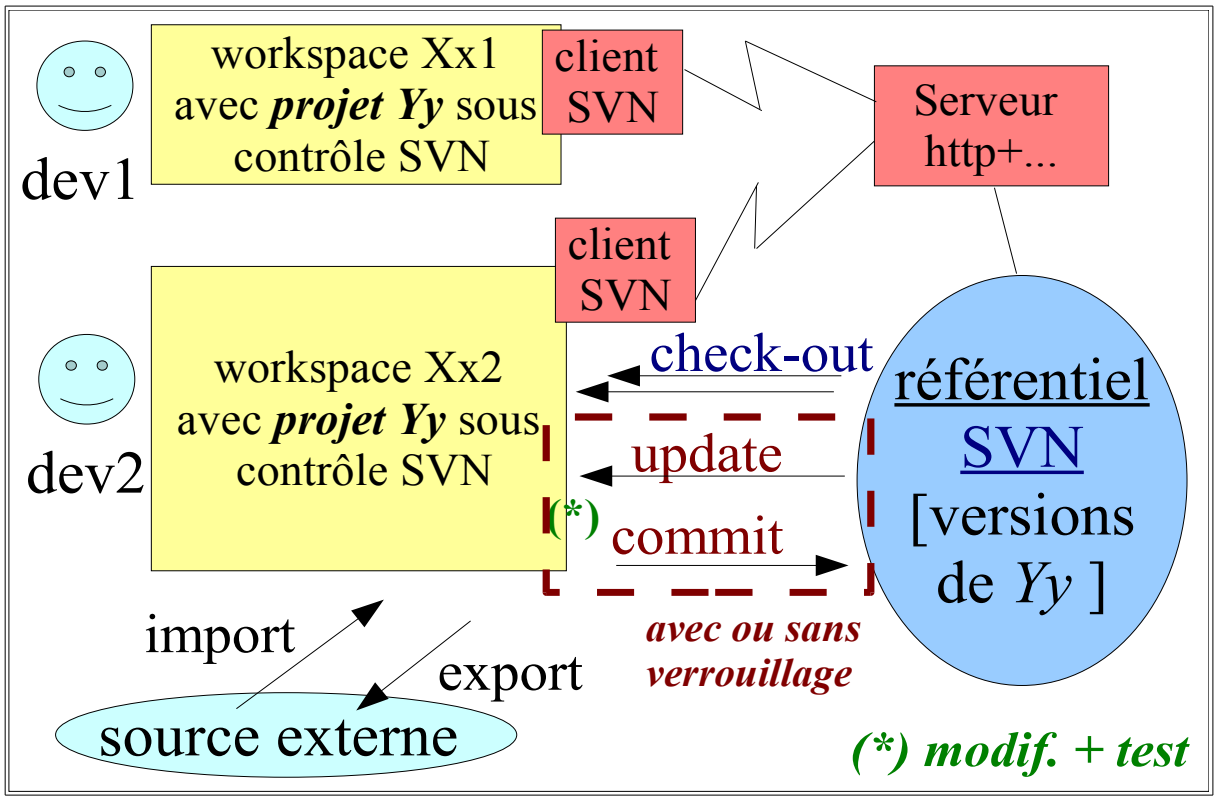
XIX - Annexe – Versions des sources (SVN,GIT)

1. Gestionnaires de code source (CVS , SVN, GIT)

1.1. Terminologie & fonctionnement de CVS et SVN

CVS = **C**oncurrent **V**ersion **S**ystem **SVN** (alias **Subversion**) = évolution de CVS .

Il s'agit d'un produit "Open Source" qui permet de **gérer différentes versions d'un ensemble de fichiers sources** lié au développement d'un certain module logiciel.
Différents programmeurs peuvent travailler en équipe sur des fichiers partagés au niveau d'un référentiel commun.



Vocabulaire utilisé par CVS:

import	Créer un nouveau module en rapatriant dans le référentiel le code d'un répertoire existant [qui devient alors "l'ancienne source"].
check-out	Récupérer une copie à jour (vue locale à un programmeur) d'un module logiciel géré par CVS. Cette vue locale correspondra à un répertoire de travail qui sera sous le contrôle de CVS [NB: Ce répertoire contiendra un sous répertoire CVS]
update	Récupérer la dernière révision d'un fichier (pour lecture et/ou mise à jour)
commit	Enregistrer une nouvelle version d'un fichier (après modification et test unitaire). CVS incrémente alors automatiquement le numéro de révision.
export	Extraire depuis le référentiel une copie d'un module. Le répertoire externe alors créé ne sera plus sous le contrôle de CVS [Là est la principale différence avec un check-out classique]

...	

Vocabulaire (suite) utilisé par CVS:

module	module logiciel (Application, Librairie, ...) à placer sous le contrôle de CVS. Un module correspond à une arborescence de fichiers (sources, make).
revision	version d'un fichier (ex: **1.1** , 1.2 , ... 1.9 , 1.10 , **1.11** , ...) NB: incrémentation automatique lors d'un "commit"
release	version d'un module logiciel complet (ex: "**rel-1-1**" , "**rel-1-2**" , ...) NB: un nom de release correspond à un "**Tag** CVS" et doit commencer par une lettre et ne doit pas contenir de point (.) ni de blanc .

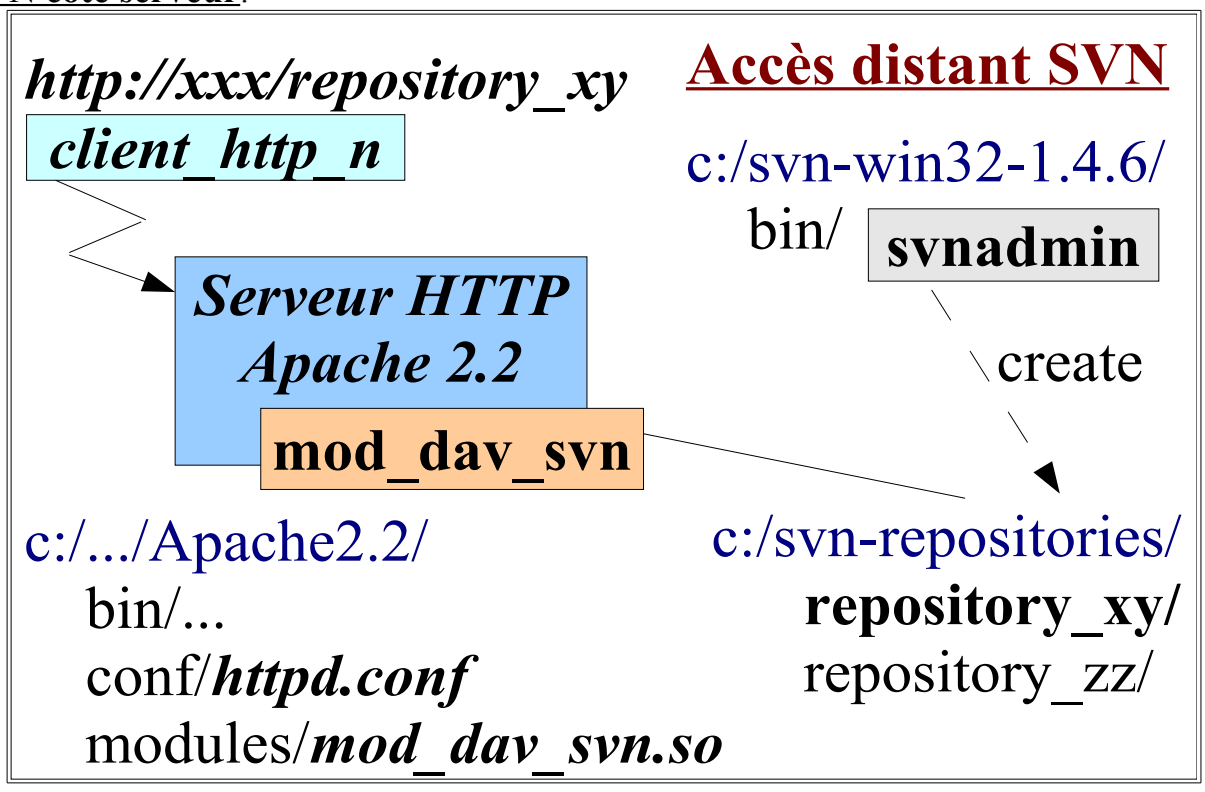
NB (pour CVS): Les fichiers du code source sont modifiés à des rythmes très différents :
un fichier *Xxx.java* peut se trouver en *révision 1.2* alors qu'un autre fichier *Yyy.java* peut se trouver au même moment en *révision 1.8* .
A un moment donné , il faut enregistrer une image complète de l'ensemble du projet/module.
Une release "rel-1-1" ou "rel-1-2" comportera donc des fichiers de code avec des numéros de révisions différents .

1.2. Présentation de SVN (Subversion)

SVN (***subversion***) est un nouvel outil de gestion de versions (beaucoup plus récent que CVS).
SVN se veut avant tout être une "***version améliorée de CVS"*** qui

- ne remet pas en cause les principes fondamentaux de CVS (référentiel commun et commit,update,...)
- a refondu l'implémentation du serveur et des référentiels (meilleur gestion des transactions, protocole d'accès plus simples, ...)

SVN coté serveur:



Principales différences entre SVN et CVS:

- Les numéros de révisions SVN sont liés à l'ensemble d'un module (projet) et non plus à un seul fichier (comme pour CVS).

- ....

2. Installation d'un serveur SVN et d'un référentiel SVN

2.1. installer SVN et créer un référentiel

- **télécharger SVN** alias subversion (site tigris ou bien Sonatype ou ...)
  - version "bin for win32 build with apache2.2" ou autre

- **installer le code du serveur SVN** en dézippant vers c:
  --> *C:\svn-win32-1.4.6* *ou autre (plus récent)*

- **créer un répertoire qui accueillera les référentiels**
  mkdir c:\svn-repositories
  et ce placer dedans (**cd**)

- **créer un référentiel vide**:
  set SVN_HOME=c:\svn-win32-1.4.6
  "%SVN_HOME%/bin/svnadmin" create myrepository

2.2. paramétrer un accès distant HTTP/DAV depuis Apache2.2

- **installer si besoin apache2.2**
- **installer le module d'accès distant dans apache2.2**:
  recopier les fichiers "***mod_dav_svn.so***" et "***mod_authz_svn.so***"
  de %SVN_HOME%/bin vers %Apache2_HOME%/modules
  recopier également les ***dll*** de %SVN_HOME%/bin
  vers "%Apache2_HOME%/bin *(sans écraser celles qui sont déjà présentes !!!)*

  et ajouter ceci dans **conf/http.conf** de apache2.2:

```
LoadModule dav_module        modules/mod_dav.so
LoadModule dav_fs_module     modules/mod_dav_fs.so
LoadModule dav_svn_module    modules/mod_dav_svn.so
LoadModule authz_svn_module  modules/mod_authz_svn.so

....

<Location /myrepository>
DAV svn
SVNPath C:/svn-repositories/myrepository
```

</Location>

rédémarrer ensuite Apache2 (stop/start) et vérifier l'accès distant au référentiel via l'url suivante: *http://localhost/myrepository*

2.3. Sécuriser l'accès distant au référentiel

Ajouter quelqueques paramètres au bloc <Location> de **httpd.conf** du serveur Apache2.2:

```
<Location /myrepository>
DAV svn
SVNPath C:/svn-repositories/myrepository
AuthType Basic
AuthName "SVN Repository"
AuthUserFile C:/svn-repositories/conf/dav_svn.passwd
Require valid-user
AuthzSVNAccessFile C:/svn-repositories/conf/authz.txt
</Location>
```

NB1:
Le fichier ***dav_svn.passwd*** comportera les ***mots de passe cryptés pour Apache2/Dav/Svn*** .

Pour créer (ou complémenter) ce fichier, il faut déclencher l'instruction suivante:
"%APACHE2_HOME%/bin/htpasswd" -c -m conf/dav_svn.passwd *svnuser*
puis saisir le mot de passe (ex: ***svnpwd***) lorqu'il sera demandé

NB2:
Le fichier ***authz.txt*** comportera les ***droits d'accès au référentiel SVN***.
En voiçi un exemple:

```
[groups]
dev = svnuser, svnuser2
[/]
anonymous = r
@dev = rw
[/trunk]
@dev = rw
[/branches]
@dev = rw
[/tags]
@dev = rw
```

NB3:
Toute modification de **httpd.conf** nécessite un redémarrage de Apache2.2.
Si le serveur HTTP refuse de démarrer , on peut débugger en mettant temporairement en commentaire certaines lignes de httpd.conf (ajout d'un # en début de ligne) : ceci permet de situer rapidement le problème.

2.4. installer le plugin SVN pour eclipse

- télécharger ***eclispe_svn_plugin_site-1.2.4.zip***
- installer ce plugin dans eclipse via:
  help / find & install / from Zipped update site

3. Utilisation directe d'un référentiel CVS ou SVN au sein d'un IDE (ex: eclipse)

Au sein de l'I.D.E. eclipse , il existe des perspectives et des vues dédiée à CVS ou SVN.

3.1. Connexion au référentiel CVS / SVN depuis eclipse

Depuis la **perspective "CVS Repository"** ou bien **"SVN Repositories"** ,
click droit / **New repository Location**

- pour SVN ==> **url=http://machine_avec_apache2_2_et_svn/myrepository**
  + compte SVN (***svnuser,svnpwd***)

3.2. Intégrer un embryon de nouveau projet dans un référentiel CVS/SVN (import SVN)

1) Sur un seule poste de développement , préparer un nouveau projet
2) Paramétrer sa structure (répertoire , packages , ...)
3) Depuis la perspective Java ordinaire , click droit / **Team / Share Project**

3.3. Charger ce nouveau projet sur les autres postes de l'équipe de développement depuis un référentiel CVS/SVN (checkout)

- Depuis un workspace vide , click droit / ***Import/ Checkout CVS project***
  ou bien **Import/ Other.../ Checkout SVN project**

3.4. Gérer les révisions CVS/SVN depuis eclipse

Depuis perspective java , click droit , **Team** , ...
... **commit** ==> pour créer une nouvelle révision (après changement dans le code + tests)
.... **show in Resource History** ==> pour obtenir la liste des révisions (et par click droit sur une ancienne révision on peut obtenir la liste des différences vis à vis de la version actuelle)

3.5. Gérer les versions / releases

Depuis perspective java , click droit , **Team** ...
.... **Tag as version (ex: Rel_1_1)** [effectuer un Refresh dans la vue ressource History]

3.6. Autres commandes utiles

- **Team / disconnect** pour se déconnecter totalement du référentiel CVS/SVN [opération inverse = import / checkout complet]
- **Replace with** => pour remplacer la révision courante par une autre (éventuellement plus ancienne)
- **Compare with** => pour afficher les différences entre plusieurs révisions

3.7. Remarques importantes

- Après avoir ***supprimé un fichier*** définitivement inutile (depuis le workspace eclipse), il faudra penser à déclencher un "Team/**commit**" sur le **répertoire parent** de façon à ce que le référentiel prenne lui aussi en compte la suppression du fichier devenu inutile.
  Et un "Team/update" sur ce même répertoire parent fera disparaître le fichier effacé sur les autres postes de développement connectés au même référentiel.
- Après avoir ajouté un nouveau fichier , on peut activer le menu "***Team/Add to Version Control***" pour que ce nouveau fichier soit vu par SVN (il fera plus tard l'objet d'un "commit").
- Si suite à une tentative de commit via SVN, une erreur survient (conflit/merge) car un autre développeur a modifié ce fichier et a déjà effectué un commit il faut alors procéder de la façon suivante:
  a) effectuer un "**Team/update**" (ce qui conduit à une vue locale de type "merge")
  b) résoudre si possible les conflits en éditant le fichier (fusionner variante, supprimer variante, ...) soit manuellement , soit via "**Team/edit conflicts**".
  c) marquer les conflits comme résolus via "**Team/Mark Resolved**" puis tester et effectuer un "**Team/commit**" ou bien si conflits trop importants effectuer un "**Team/Revert**".
- Le menu "***Team/Synchronized with Repository***" permet de visualiser l'ensemble des différences entre la vue locale et le référentiel (avant les commit & update).

4. GIT : nouvelle technologie pour contrôler le code source

Après **CVS** et **SVN** , est récemment apparue la technologie "**GIT**" qui est :

- plus performante (référentiel plus petit)
- utilisable en mode distribué
- plus souple (vis à vis d'un passage d'une branche à une autre)

Les fonctionnalités fondamentales apportées par GIT sont cependant les mêmes que celles de SVN

==> même principes d'utilisation depuis eclipse via un "***plugin GIT pour Eclipse***" que l'on trouve de façon bien intégré à partir de eclipse 3.7 (indigo).

XX - Annexe – énoncés des TP

Les TP ci-après ne sont que des propositions (à caractère indicatif et non impératif).
Il ne faut pas hésiter à tester tout un tas de variantes (selon ses préférences personnelles).

NB: chaque nouvelle classe développée dans un TP devra être placée
dans un **package** adéquat (dont le nom est à choisir) .

1. TP1 (prise en main du jdk)

Objectif : Edition, Compilation et Exécution sans eclipse , avec un simple éditeur de texte et le jdk :

A faire : Hello World !!!

2. TP2 (première classe simple, conventions JavaBean)

Objectif : Ecrire une classe Java dans les règles de l'art

A faire:

- Créer un nouveau projet java de nom "***tpInit***" sous eclipse en demandant une séparation entre le répertoire des fichiers sources "**src**" et le répertoire (output) des fichiers compilés "**bin**".
- Créer la classe "**ConsoleApp**" avec une méthode **main**() qui servira aux tests.
- Créer (et tester) une première version d'une classe "**Personne**" avec des attributs "**nom**", "**age**", "**poids**" déclarés provisoirement "public".
- Coder une méthode "public void **afficher**()" qui affiche à l'écran les valeurs des attributs. (+ tests)
- Rendre "**private**" les **attributs** de la classe "Personne"
- Générer les méthodes **get**Xxx()/**set**Xxx(...) (+tests)
- Coder quelques **constructeurs**. (+tests)
- Coder la méthode classique "public String **toString**()" (provenant de la classe Object) en y construisant une chaîne de caractères complète regroupant tous les attributs + return
- Reprogrammer la méthode afficher() de façon à ce quelle appelle toString() en interne.
- Créer deux instances p1 et p2 de la classe Personne avec les mêmes valeurs internes.
- Comparer ces 2 instances et afficher si (oui ou non) les valeurs internes sont identiques.
- Reprogrammer la méthodes "public boolean **equals**(Object obj)" sur la classe Personne. Cette méthode doit renvoyer "true" si et seulement si toutes les valeurs internes de this et de obj sont identiques.

3. TP3 (classe "AvionV1" avec tableau de "Personne")

Créer une classe appelée "AvionV1" qui comportera :

- un attribut "***tabElements***" qui sera une référence sur un futur tableau de Personnes

  La taille maximum de ce tableau sera fixe (ex: 50)
- un attribut "***nbElements***" qui comptabilisera le nombre d'éléments insérés dans le tableau (pas obligatoirement rempli).
- Une méthode ***addElement***() permettant d'ajouter une référence d'élément dans le tableau interne
- Une méthode ***initialiser***() qui créera quelques Personnes et ajoutera les références au tableau.
- Une méthode ***afficher***() qui affichera tous les éléments de l'avion.

4. TP4 static – constante , ...

- Déclarer une constante ***AvionV1.TAILLE_MAX*** correspondant à la taille maxi du tableau de l'avion
- Ajouter les éléments suivants sur la classe "Personne" :
  * variable de classe privée "nbInstances"
  * méthode de classe getNbInstances() (pas de set)
  * constructeur(s) incrémentant nbInstances
  * méthode finalize() décrémentant nbInstances
- Depuis la méthode main() , appeler plusieurs fois la méthode getNbInstances() sur la classe Personne (avant et après avion = null; System.gc(); par exemple).
- Déclencher le calcul racine carré de 81 dans la méthode principale main().

5. TP5 (classe "Employe" héritant de "Personne")

- Créer et tester une sous classe "**Employe**" (héritant de "Personne") et comportant un **salaire** en plus.
- Redéfinir la méthode **toString**() sur la classe "Employe" en y effectuant un appel au **toString**() de la classe "Personne" et en concaténant le salaire en plus.
- Bien soigner l'écriture des différents constructeurs .
- Retoucher la méthode **initialiser()** de AvionV1 de façon à créer et ajouter quelques Employés dans le tableau interne (en plus des Personnes déjà placées).
- Vérifier la **polymorphisme** s'effectuant automatiquement au sein de afficher() / toString() sans avoir à reprogrammer quoi que ce soit.

6. TP6 (classe abstraite "ObjetVolant")

- Créer une nouvelle classe abstraite "**ObjetVolant**" comportant un attribut privé "couleur" de type String et les méthodes traditionnelles getCouleur() / setCouleur(...).
  Cette classe comportera une méthode abstraite **getPlafond**() prévue pour retourner l'altitude maximale que l'objet volant est capable d'atteindre .
- Retoucher la classe AvionV1 de façon à ce quelle hérite maintenant de la classe abstraite ObjetVolant .

7. TP7 (interface "Descriptible" ou "Transportable")

- Créer une interface "**Descriptible**" ou "***Transportable***" comportant les méthodes "**getDesignation**() et **getPoids**()"
- Remodeler la classe "Personne" de façon à ce qu'elle implémente les méthodes de l'interface "**Descriptible**". [ex: getDesignation() peut appeler toString()]
- Créer une nouvelle classe "**Bagage**" (avec label, poids, volume) qui implémente l'interface "Descriptible" ou "Transportable".
- Effectuer une copie AvionV2 à partir de la classe AvionV1.
- La nouvelle variante AvionV2 comportera maintenant un tableau de référence sur des éléments quelconques de type "Descriptible" ou "Transportable".
- La nouvelle version de la méthode afficher() de AvionV2 pourra par exemple afficher les désignations de chacun des éléments et calculer la charge complète de l'avion (somme des poids des éléments).

8. TP8 (Exception):

Ecrire une toute petite application qui calculera la racine carrée du premier argument passé au programme.
Utiliser des traitements d'exception pour gérer les cas anormaux suivants:

- appel du programme sans argument ==> Array Index Out Of Bound Exception
- argument non numérique ==> Number Format Exception
- ...

V1 : simple try/catch dans main() sans if
V2: le main délègue le calcul à un objet de type "***SousCalcul.java***" (à programmer).
La méthode "***public double calculerRacine(double x)***" de SousCalcul devra tester si x est <0 et devra dans ce cas remonter une exception de type "***MyArithmeticException***" (à programmer en héritant de Exception ou RuntimeException et avec un constructeur de type:

MyArithmeticException(String msg) { super(msg); } .

Eventuelle V3 (facultative) : niveau intermédiaire (Calcul) entre le main() et SousCalcul.

NB (Sous eclipse): Après un premier lancement de l'application via "*click droit/Run as/java application*", un paramétrage de la partie "*Prog. arg*" de l'onglet "***arguments***" de "***Run/Run ...***" permet de préciser un (ou plusieurs) argument(s) de la ligne de commande/lancement [ex: 81 ou a9 ou rien]

9. TP9 (Collections & Generics)

main() à retoucher pour tester ==> AvionV1 , AvionV2 , AvionV3, AvionV4

Partie1 du TP:

Créer une nouvelle version "**AvionV3**" par copie de "AvionV2"

Remplacer le tableau interne "tabElements" et "nbElements" par une **Collection** "***listeElements***" comportant entre autres la méthode prédéfinie "*size*()".

Retoucher le code interne de addElement() et afficher() pour que tout soit cohérent.

Partie2 du TP:

Créer une nouvelle version "**AvionV4**" par copie de "AvionV3"

Introduire la syntaxe des ***generics*** du jdk>=1.5 [.... <Descriptible> ou ...<Transportable>]

Utiliser la nouvelle boucle **for** (au sens "forEach") dans la méthode *afficher*().

10. TP10 (Dates & ResourceBundle)

Insérer le fichier "**MyResources.properties**" dans le code source de l'application avec le contenu suivant:

msg.day=day

mas.month=month

msg.year=year

Développer ensuite une version française (_fr) avec "année" , "mois" et "jour".

Dans une sous méthode "*static void test_dates()*" appelée par main() effectuer les tâches suivantes:

- Récupérer les valeurs des messages "msg.day" , "msg.month" et "msg.year" .
- Récupérer les valeurs entières day , month, et year depuis la date d'aujourd'hui.
- Afficher proprement un message de type "annees: 2007, mois: 2 , jour: 12 "
  ou "year: 2007, month: 2 , day: 12 " via System.out.printf()

11. TP11 (Application *ou Applet* Dessin en awt/swing):

Ecrire une application ou un applet "Dessin" capable de dessiner des lignes, des rectangles ou des cercles avec une couleur que l'on choisira dans une liste déroulante.

phase1 --> générer la classe de la fenêtre principale (ou de l'applet et sa page html).
phase2 --> coder la structure graphique (imbrication de composants, layout)
phase3 --> coder les événements appropriés.

12. TP12 (Gestion des fichiers) :

Partie1 : Ecrire une application comportant (d'une façon ou d'une autre)
une instance d'une classe "JPanelPays" que l'on fabriquera.
La classe "JPanelPays" héritera de "javax.swing.JPanel" et comportera
une "JList" (imbriquée dans un JScrollPane") permettant d'afficher une
liste de pays que l'on récupèrera dans un fichier texte (c:\stage\
ressources\data_files\listePays.txt).
Conseil : remonter en mémoire les données dans un vecteur d'objets "Pays"
(nouvelle petite classe avec attributs "nomPays" et "capitale" et méthode toString()).
On pourra éventuellement développer un jeu consistant à trouver le pays
correspondant à une capitale sélectionnée aléatoirement.

Partie2 : Reprendre le Tp "ConsoleApp / AvionV2" et y ajouter du code permettant
de déclencher une sérialisation complète des données d'un "AvionV2".
--> rendre tout "Serializable" via des "implements" qui vont bien.
--> relire le fichier généré et remonter (via xxx.readObject()) les
données dans une seconde instance que l'on affichera à l'écran.

13. TP 13 (Accès aux bases de données) :

Ecrire (ou agrandir) une application comportant (d'une façon ou d'une autre)
une instance d'une classe "JPanelGeo" que l'on fabriquera.
La classe "JPanelGeo" héritera de "javax.swing.JPanel" et comportera
un "JTree" (imbriqué dans un JScrollPane") permettant d'afficher une
liste de régions et de départements que l'on récupèrera dans une
base de données (c:\stage\ressources\database\access\geo.mdb).
Conseil : remonter en mémoire les données dans des tableaux ou collections d'objets
"Departement" et "Region" (nouvelle petites classes avec attributs "nom" et "prefecture"
"..." et méthode toString()). Ces petits objets pourront également correspondrent à
la partie interne des noeuds de l'arbre (DefautMutableTreeNode du TreeModel).
On pourra éventuellement développer un jeu consistant à trouver le département
correspondant à une préfecture sélectionnée aléatoirement.

14. TP 14 (Gestion des threads) :

(dans un nouvel onglet "*OngletThread*")

- Partie 1 ==> démarrer en // 5 nouveaux threads qui afficheront "1" puis "2" puis ... puis "20" dans une zone graphique réservée (propre à chacun des threads).

- Partie 2 ==> développer une classe "PoolDeRessource" permettant d'obtenir et de libérer des ressources (ex: Objet "String"pour simulation).
  Ce pool (que l'on limitera volontairement à 3 ressources) sera ensuite utilisé par les 5 threads qui afficheront le nom de la ressource obtenue et non plus la valeur d'un simple compteur.