

# Java 8,11,17

## (avec JDBC)

### Table des matières

<b>I - Présentation du langage JAVA.....</b>	<b>6</b>
1. Historique et évolution.....	6
2. Machine virtuelle JAVA.....	8
3. Principales particularités du langage JAVA.....	10
4. Structuration des API Java (JavaSE / JavaEE).....	12
5. IDE (Environnement de développement intégré).....	14
6. Maven (ou gradle) quasi indispensable.....	14
<b>II - Éléments de base du langage Java.....</b>	<b>16</b>
1. Compilation , exécution et IDE.....	16
2. Types de données.....	18
3. Classes, instances et références.....	22
4. Ramasse miettes (G.C.).....	29
5. Opérateurs du langage Java.....	31
6. Boucles & instructions de Java.....	34

7. Chaînes de caractères.....	35
8. Tableaux.....	38
9. Méthode et variables de classes.....	41
<b>III - Héritage , polymorphisme , interface.....</b>	<b>44</b>
1. Généralisation / Héritage.....	44
2. Polymorphisme.....	49
3. Classes abstraites.....	51
4. Interfaces.....	53
<b>IV - Eléments structurants de java.....</b>	<b>56</b>
1. Packages et archives (.jar).....	56
2. Gestion des exceptions.....	63
3. Présentation des API de log.....	67
4. Propriétés du système.....	70
5. Classes imbriquées (depuis jdk 1.1).....	71
6. Collections (depuis le jdk 1.2).....	72
7. Generics (depuis Java 5).....	76
8. Modules java 9+ (essentiel).....	82
<b>V - Prog. Fonctionnelle (lambda, stream, ... ).....</b>	<b>88</b>
1. Java multi-paradigme "objet + fonctionnel".....	88
2. Lambda expressions et prog. fonctionnelle.....	90
3. Streams (java 8 et plus).....	97
<b>VI - Classes utilitaires , aspects divers.....</b>	<b>101</b>
1. Eléments de Java >=5 (jdk 1.5 , 1.6 et 1.7).....	101
2. Quelques classes utilitaires.....	103
3. Internationalisation.....	105
4. Mise en forme du texte (java.text).....	106
5. Optional , DateTime , .....	107
<b>VII - Entrées/sorties (io,nio,nio2) – fichiers.....</b>	<b>112</b>
1. Essentiel de java.io.....	112
2. Spécificités à connaître et détails intéressants.....	117
3. Java.util.Scanner (depuis Java 5).....	117
4. try with resources (AutoClosable).....	118
5. NIO2 (new IO).....	120

6. Principales classes et interfaces de nio2.....	120
7. Gestion des chemins (Path).....	122
8. Classe utilitaire Files (" <i>helper</i> " avec 50 méthodes statiques).....	126
9. Parcours des éléments d'un répertoire.....	128
10. Parcours d'une hiérarchie de répertoires (visiteur).....	130
11. FileSystem (par défaut et "personnalisé").....	130
12. Lecture et écriture dans un fichier.....	131

## VIII - Introspection et Sérialisation..... 133

1. Introspection (java.lang.reflect).....	133
2. Sérialisation (et persistance élémentaire).....	137

## IX - JDBC (accès aux bases de données)..... 140

1. JDBC : Présentation et structure.....	140
2. Paramétrage et établissement d'une connexion.....	141
3. Principaux objets de l'api JDBC.....	143
4. Lancer un ordre sql.....	143
5. Effectuer une requête (select).....	144
6. Balayer les lignes du résultat.....	144
7. Accès à la structure de la base (MetaData).....	145
8. Gestion des transactions (tout ou rien).....	146
9. Préparer et lancer n fois un ordre Sql paramétrable.....	146
10. Appels de procédures stockées.....	146
11. Astuce pour fermer proprement les connexions.....	148
12. Récupérer la valeur d'une clef auto-incrémentée.....	148
13. Fonctionnalités à partir de la version 2 de JDBC.....	149
14. Mémento SQL + Mise en oeuvre MySQL.....	150

## X - Threads (java)..... 152

1. Concept de threads.....	152
2. Gestion des threads avec java.....	153
3. Synchronisation des threads.....	155
4. Attente et rendez-vous ( wait & notify ) java 1.0.....	157
5. Concurrent api.....	158
6. ForkJoin.....	170
7. CompletableFuture , streams asynchrones.....	179

## XI - Api pour IHM/GUI (swing, ...)..... 187

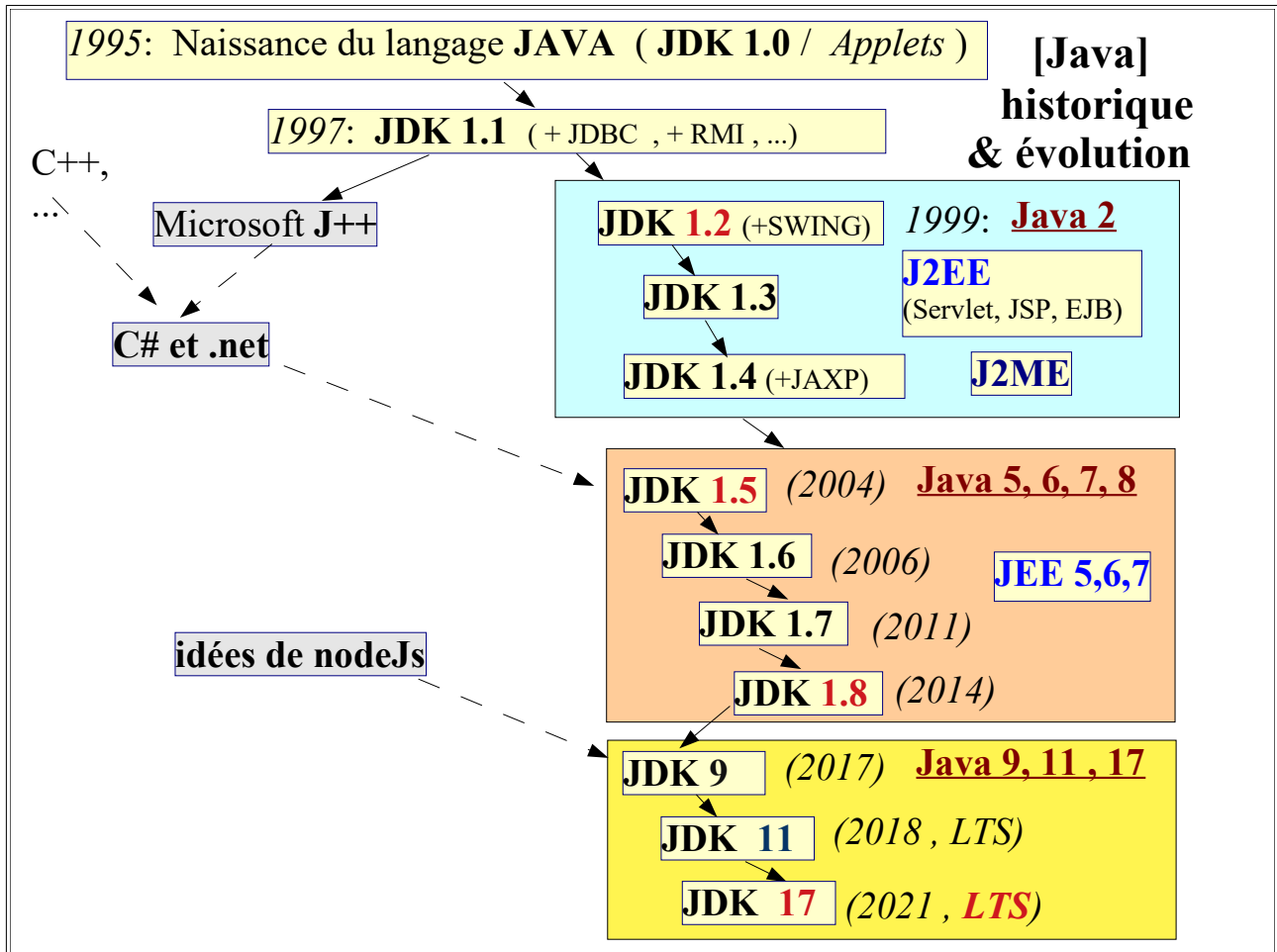
1. Eléments de base sur AWT/SWING et événements.....	187
2. Gestion des événements.....	194
<b>XII - Annexe – nouveautés v12-17.....</b>	<b>199</b>
1. Switch expressions.....	199
2. Pattern Matching instanceof.....	201
3. TextBloc.....	202
4. Record (classe de données simplifiée pour DTO).....	204
5. Sealed classes.....	209
6. Pattern matching sur switch/case of object.....	211
7. utilitaire jpackage.....	212
8. Autres apports des versions récentes (12,...,17).....	213
9. Nouveau type de licence depuis java 17.....	213
<b>XIII - Annexe – détails sur les modules.....</b>	<b>214</b>
<b>XIV - Annexe – Quelques API de java.....</b>	<b>226</b>
1. Process Api.....	226
<b>XV - Annexe – Tests unitaires (JUnit 3 , 4 et 5).....</b>	<b>230</b>
1. Tests unitaires avec JUnit (3 ou 4).....	230
2. Tests unitaires avec JUnit 5.....	237
<b>XVI - Annexe – Annotations Java.....</b>	<b>245</b>
1. Annotations : Présentation et intérêts.....	245
2. Annotations et méta-annotations prédéfinies.....	246
3. Création de nouvelles annotations.....	247
4. Insertion d'annotations au sein d'un code source.....	248
5. Analyse et traitement des annotations via l'utilitaire APT (Annotation Processing Tool).....	249
6. Accès aux annotations (de rétention RUNTIME) via l'introspection de java 5... 252	
<b>XVII - Annexe – Structure globale appli.....</b>	<b>253</b>
1. Architecture généralement recommandée.....	253
<b>XVIII - Annexe – énoncés des TP.....</b>	<b>254</b>
1. TP1 (prise en main du jdk).....	254

---

2. TP2 (première classe simple, conventions JavaBean).....	254
3. TP3 (classe "AvionV1" avec tableau de "Personne").....	255
4. TP4 static – constante , .....	255
5. TP5 (classe "Employe" héritant de "Personne").....	255
6. TP6 (classe abstraite "ObjetVolant").....	256
7. TP7 (interface "Descriptible" ou "Transportable").....	256
8. TP8 (Exception):.....	256
9. TP9 (Collections & Generics).....	256
10. TP10 (Dates & ResourceBundle).....	257
11. TP11 (Application <i>ou Applet</i> Dessin en awt/swing):.....	257
12. TP12 (Gestion des fichiers) :.....	257
13. TP 13 (Accès aux bases de données) :.....	258
14. TP 14 (Gestion des threads) :.....	258

# 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)*.

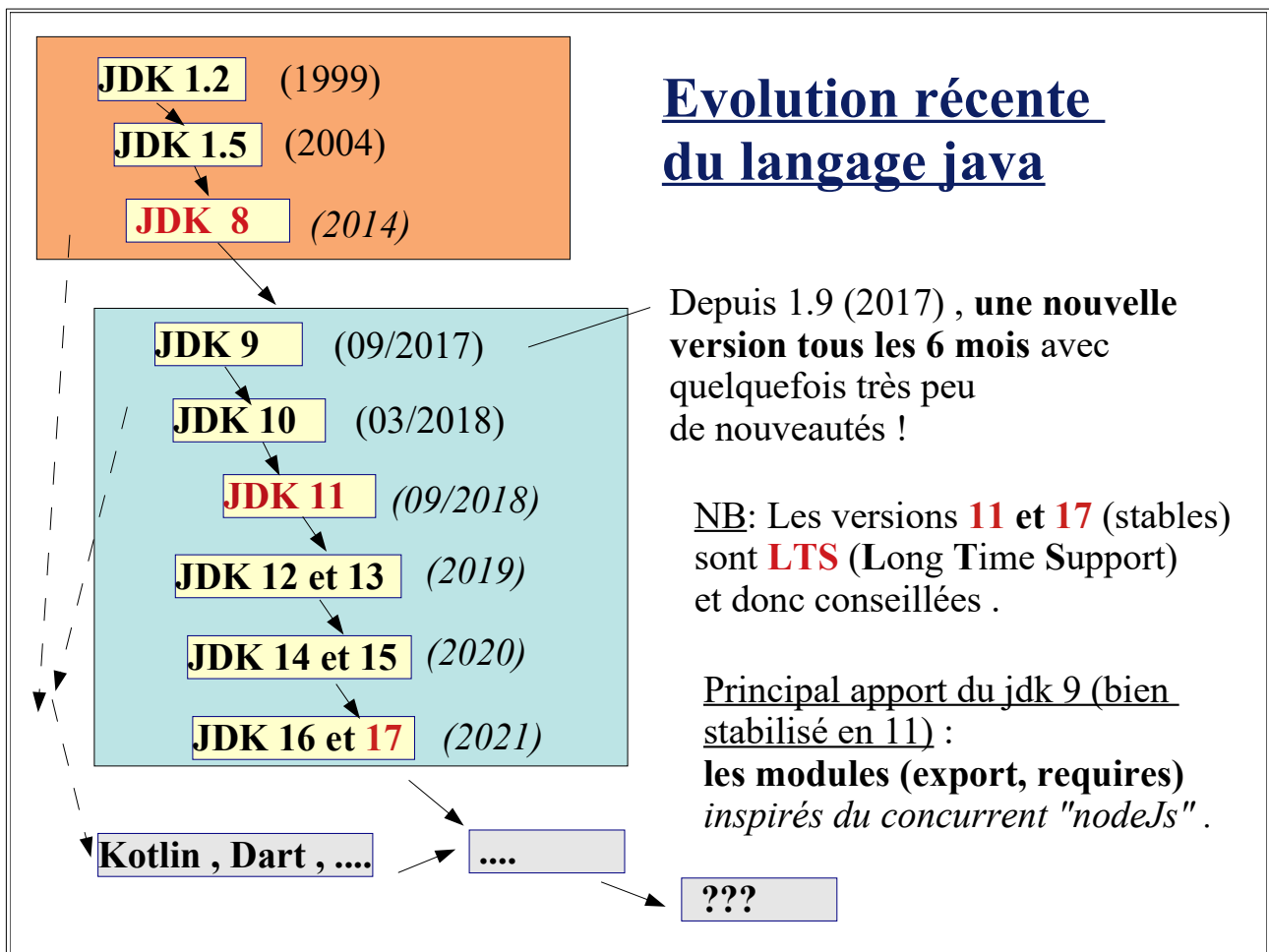
Bien que différents et incompatibles les langages Java et C# comportent beaucoup de points communs (syntaxe et architecture assez proches).

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**.



NB : à partir du jdk 1.11 , java-fx est devenu une extension facultative (désormais plus incluse dans le jdk ) .

NB : a coté du jdk officiel (de l'entreprise Oracle) , il existe également le "**openJdk**" entièrement "open source" (un peu moins complet que le jdk d'Oracle mais sans partie potentiellement payante en production) .

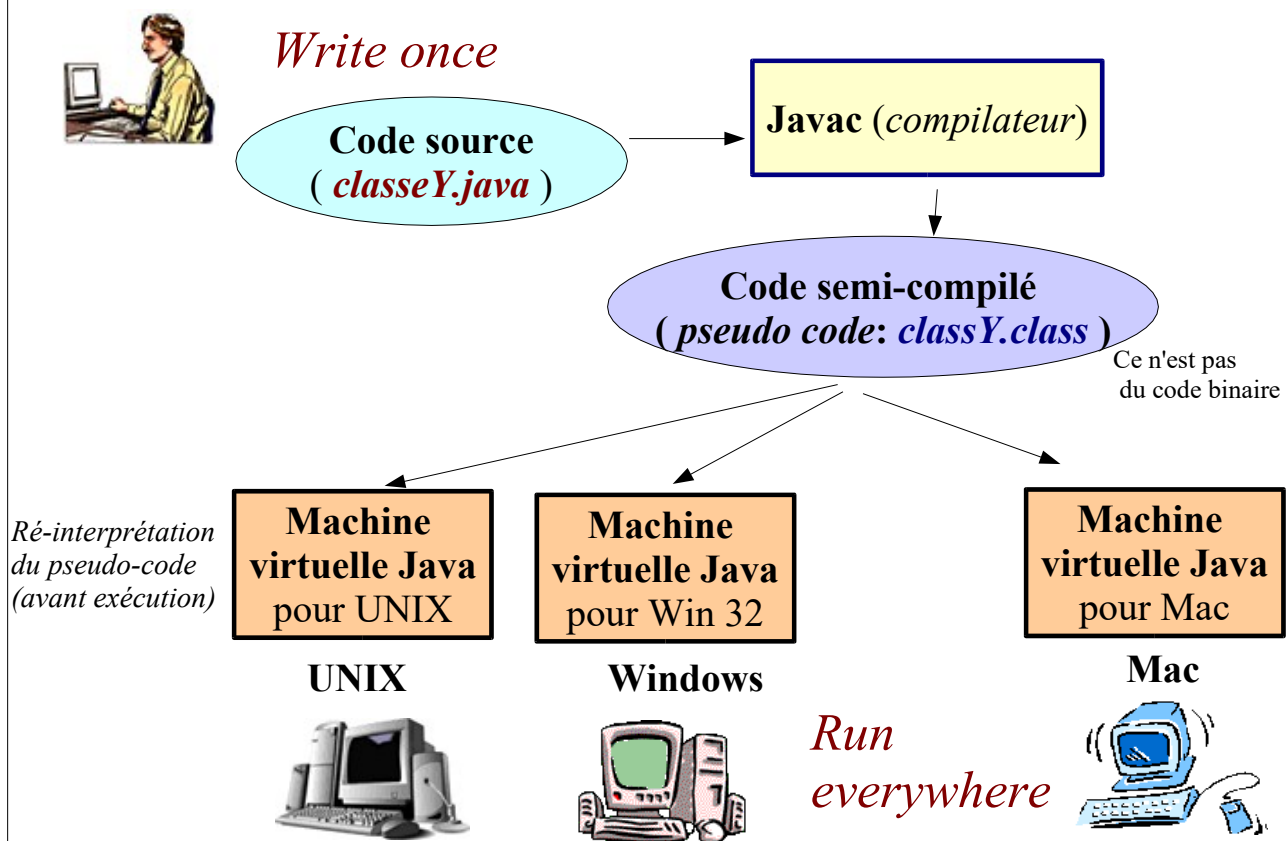
## 2. Machine virtuelle JAVA

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

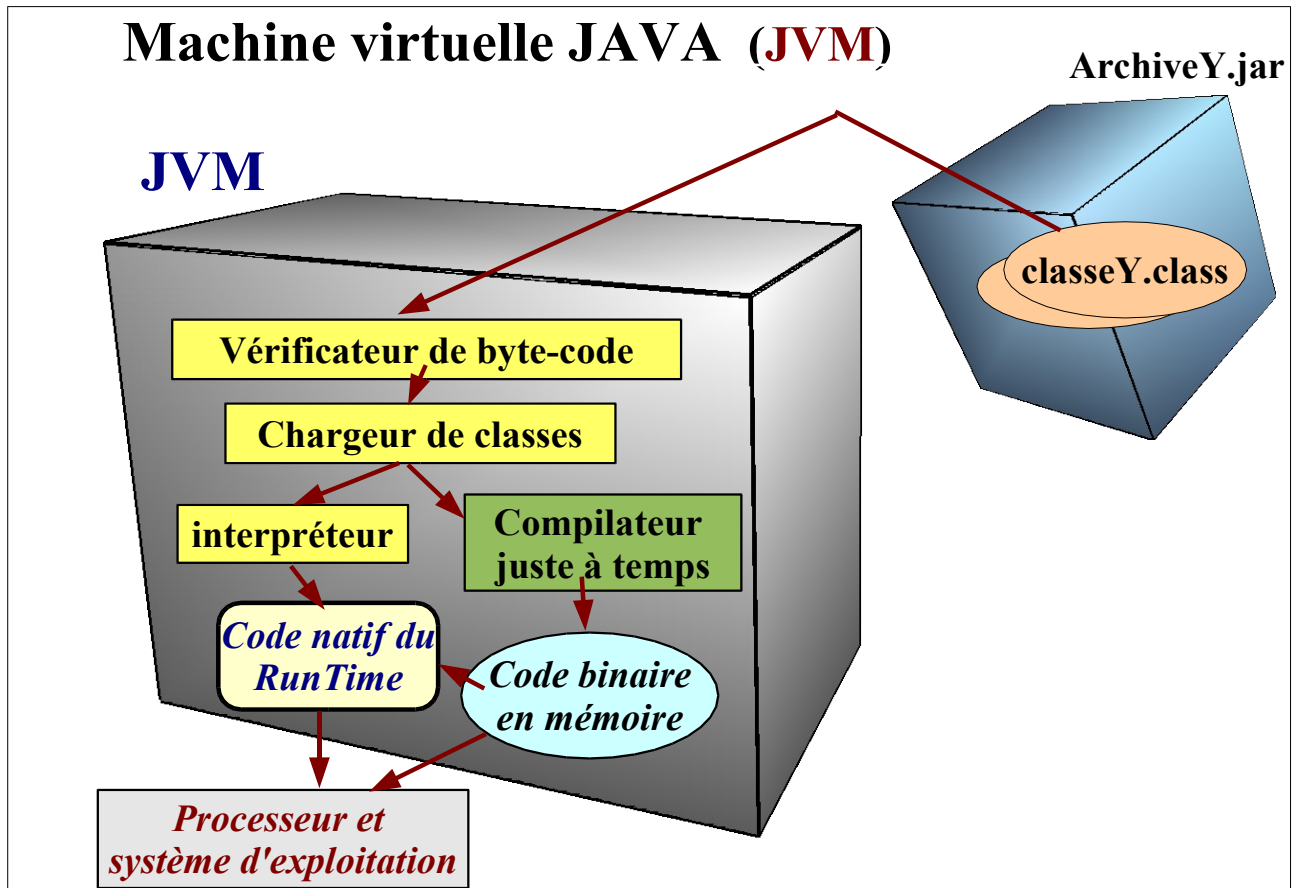
- 1) Le **développeur écrit du code source** (*classeY.java*) **puis le compile sur n'importe quelle sorte de machine** (Windows , Unix , Linux , ....).
- 2) Cette **compilation partielle** ne génère pas un code binaire compréhensible que par un certain type de machine mais génère un **pseudo-code portable** 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*) .

**Java: langage *partiellement compilé* et *interprété* (via JVM)**





**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**.

==&gt;

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

<b>Langage générique</b> <i>[ large palette d'applications ]</i>	<p><u>Le langage Java peut servir à créer différentes sortes d'entités:</u></p> <ul style="list-style-type: none"> <li>♦ <b>Applications autonomes</b> (ne nécessitant qu'une JVM)</li> <li>♦ <b>JavaBean</b> = composants quelconques (pour composer une interface graphique ou pour effectuer des traitements "métier").</li> <li>♦ <b>Servlet</b> = composant permettant de générer des pages WEB (<i>Un servlet s'exécute coté serveur</i>)</li> <li>♦ <del><b>Applet</b> (mini-application s'exécutant coté client, l'affichage s'effectue dans une sous fenêtre du navigateur internet)</del></li> <li>♦ <b>Application native android</b> , ...</li> </ul>
<b>Complètement orienté objet</b>	<p>Java est un langage résolument <b>orienté objet</b> .</p> <p>Les fonctions globales n'existent pas en java , <i>toutes les fonctions sont obligatoirement intégrées dans des objets</i> .</p> <p>Offrant un support à tous les principaux concepts objets (classe, instance, encapsulation, polymorphisme , héritage , ... ) , Java permet de <b><u>très bien structurer les programmes</u></b> (==&gt; <b>bonne modularité</b>).</p>
<b>Relativement simple</b>	<p>La <b>gestion des références</b> (qui sont plus simples à manipuler que les pointeurs) est <b>en grande partie automatisée</b>: <i>un ramasse miette libère automatiquement les blocs mémoires devenus inutiles</i>.</p> <p>Bien que <i><b>syntactiquement proche du C++</b></i> , java ne s'est inspiré que des éléments fondamentaux et simples de ce langage .</p>
<b>Souple, expressif et bien adapté au couches hautes</b>	<p>Souple , modulaire et étant doté de beaucoup d' API prédéfinies , Java est un <b>langage de prédilection pour les développements "3-tiers"</b> liés à l'informatique de gestion et à internet.</p>

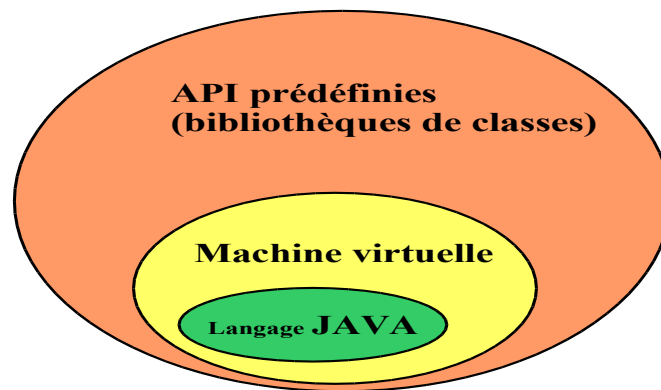
<b>Vitesse d'exécution correcte et interfaçage possible avec le langage C</b>	<p>Par contre, JAVA n'est (pour l'instant) pas du tout approprié pour coder des couches de bas niveau:</p> <p><b>Les langages C et C++ restent incontournables pour coder des traitements pointus devant s'exécuter très rapidement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

## 4. Structuration des API Java (JavaSE / JavaEE)

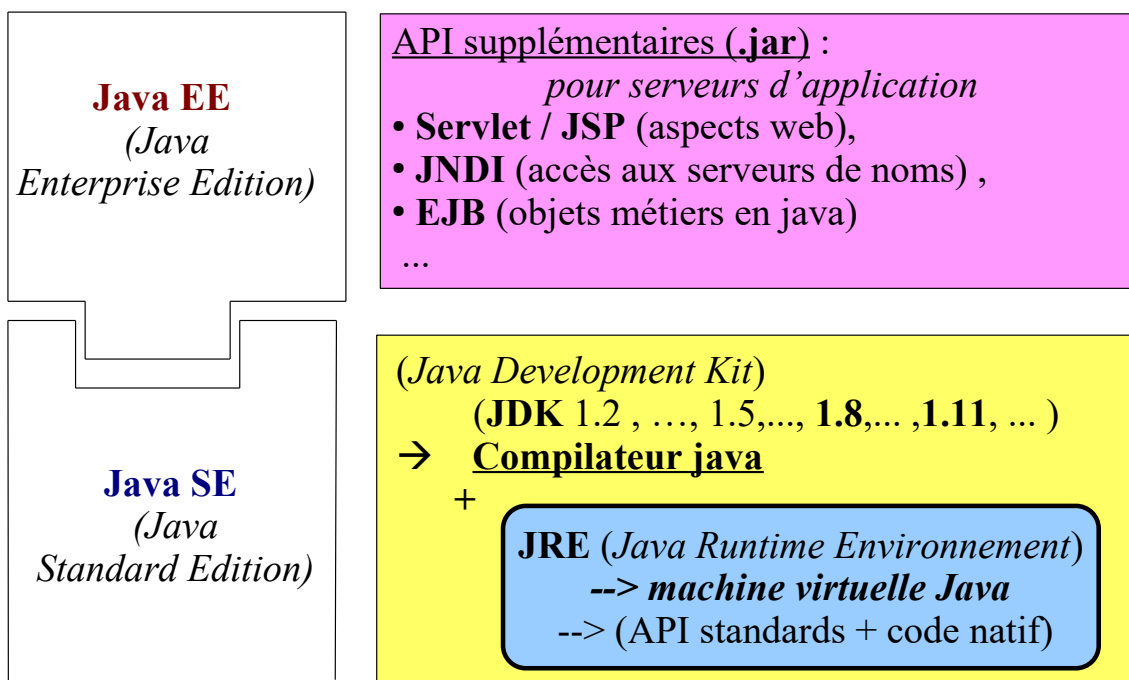
Le langage **JAVA** est associé à une machine virtuelle qui le rend portable sur une large palette de plate-formes (UNIX, Windows , ....) et est **accompagné d'un immense ensemble d'API (bibliothèques de classes) qui sont:**

- **standards** (officialisées par SUN/Oracle , JEE , ...)
- **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**



NB: il existe aussi **JavaME (Micro Edition)** = **JavaSE simplifié/allégé** pour les **cartes à puce** (ex : carte bancaire) .

## 4.1. Principales API de JAVA

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

NB : à coté des spécifications officielles "JavaEE" , il existe un très bon framework "**Spring**" (standard pas officiel mais standard de fait --> écosystème java très complet) .

## 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 **IntelliJ** ( *de JetBrains*) sont actuellement les deux **IDE Java** qui sont les plus connus et utilisés .

- **Eclipse** est complètement open source mais est quelquefois lent si trop de plugins.
- **IntelliJ** existe en 2 versions : "**community**" (gratuite) et "**ultimate**" (payante)

**NetBeans** (*de Sun/Oracle*) existe également mais est un peu moins utilisé.

**NB** : L'IDE "**Visual Studio code**" (à la base prévu pour les langages javascript et typescript) , peut également assez bien prendre en charge un projet java via une **extension (plugin)** pour "**java**" .

## 6. Maven (ou gradle) quasi indispensable

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 .

**NB** : en structurant un projet java sur la technologie "maven" (et en plaçant le code source dans un référentiel GIT) , on fait un peu comme tout le monde mais surtout , un tel projet peut être partagé (dans le cadre d'un travail d'équipe) par plusieurs développeurs qui peuvent éventuellement utiliser des IDE différents.

Autrement dit, un projet java au format "maven" (avec l'encodage "UTF-8" pour les caractères accentués) est utilisable partout (windows, linux, mac, ...) et avec un IDE quelconque (Eclipse, IntelliJ , ....) .

**Création d'un nouveau projet maven sous eclipse :**

**File/new/project.../maven/maven project...**

choisir "**Create a simple project (skip archetype selection)**"

**group id** : *tp* ou *com.xyz*   **artifactId** : *tpJava* ou autre

**Création d'un nouveau projet maven sous *intelliJ*:**

**File/new/project... / Maven , Next** (ne pas choisir d'archetype) , **renseigner le nom du projet** (et éventuellement le groupId)

***pom.xml***

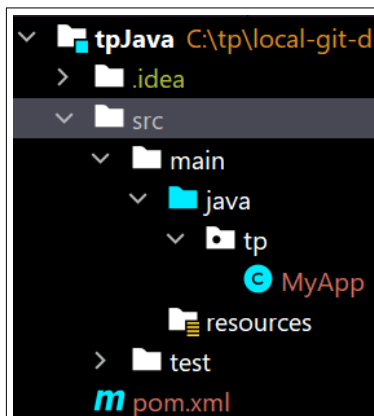
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>tp</groupId>
  <artifactId>basesJava</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.release>17</maven.compiler.release>
    <maven.compiler.target>17</maven.compiler.target>
    <maven.compiler.source>17</maven.compiler.source>
  </properties>

  <build>
    <finalName>${project.artifactId}</finalName>
  </build>
</project>
```

NB :

- *src/main/java* est un *répertoire d'organisation du code* quasi imposé par la technologie *maven*  
*src/main/resources* sert à placer d'éventuels **fichiers de configuration** (*.properties*, ...)  
*src/test/java* sert à placer les tests unitaires (selon JUnit ).
- Au sein de *src/main/java* , il faut créer un ou plusieurs **packages** (ex : *com.xyz* ou *tp* )
- Au sein du package *tp* (dans *src/main/java/tp* ) , on peut créer une classe java principale (ex : *MyApp.java* avec la méthode *main()*)

***ProjetXXX***

```
src/main/java
└─ package1 ── ClasseX.java

bin (ou target/classes)
└─ package1 ── ClasseX.class
```

## 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: MyApp*)
- Nom de méthode (fonction) commençant toujours par une minuscules (*ex: println()*)
- Nom de constante entièrement en majuscules (*ex: Math.PI*)

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

Exécution directe du code sous eclipse : click droit puis "**run as .../ java application**"

Exécution directe du code sous intelliJ: "**Run**" (triangle vert)



## 1.2. Compilation et exécution avec le jdk

Après s'être placé à la racine d'un projet (potentiellement basé sur maven),...

*lancerCompilationAvecJdk.bat*

```
set JAVA_HOME=C:\Program Files\Java\jdk-17
REM set JAVA_HOME=C:\Program Files\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.2.v20220201-1208\jre
set PATH=%JAVA_HOME%\bin
cd "%~dp0"
cd src/main/java
javac -d ../..../bin tp/MyApp.java
pause
```

l'option **-d** permet de choisir le répertoire où sera placé le code compilé (ex : "bin" ou autre)

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

*lancerAppliAvecJdk.bat*

```
set JAVA_HOME=C:\Program Files\Java\jdk-17
REM set JAVA_HOME=C:\Program Files\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.2.v20220201-1208\jre
set PATH=%JAVA_HOME%\bin
cd "%~dp0"
cd bin
java tp.MyApp
pause
```

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

## 1.3. Compilation (build) via maven

Avec maven installé (en ligne de commande) :

**mvn compile**    ou bien    **mvn install**  
ou bien    **mvn clean package**

→ le code construit est placé dans le répertoire **target** (et sous répertoire **classes**)

Déclenchement d'un build maven sous IntelliJ :

développer le panneau "**maven**" (coté droit) , développer "**Lifecycle**" , choisir "**package**" et "**Run**"

Déclenchement d'un build maven sous eclipse :

se placer sur le projet (project explorer) , click droit puis "**Run as**" / "**maven install**" ou "**maven build...**" puis goals : "**clean package**"

## 2. Types de données

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

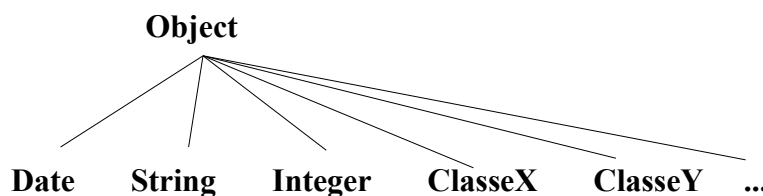
- types élémentaires (non orientés objet et entièrement orthographiés en minuscule) :

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

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

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

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



Un "*ArrayList*" 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 <sup>E</sup> -45 à 3.40282347 <sup>E</sup> +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 '\u0041' 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**
- **0xa25c** 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);
```

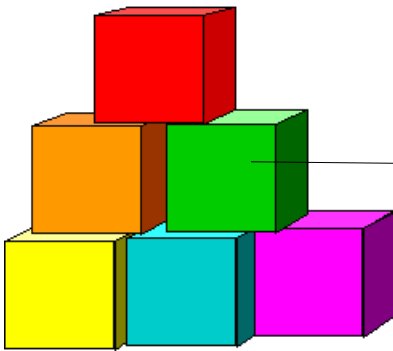
```
ch=String.valueOf(b);   bb=Boolean.valueOf(ch).booleanValue();
```

```
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
```

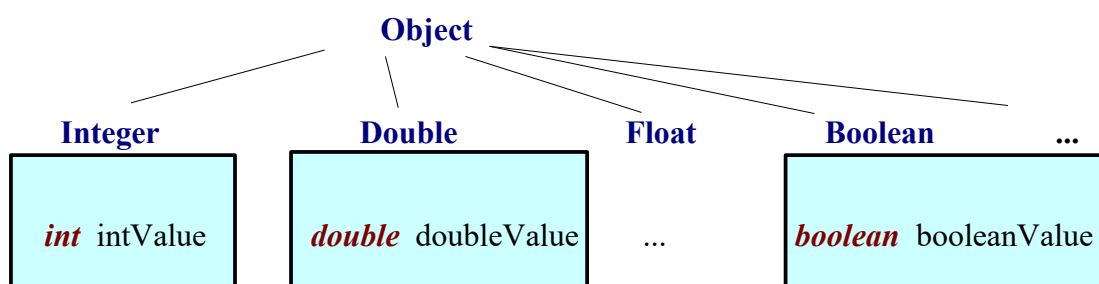
```
ref = new ClasseY(); // création d'un nouvel objet (instance) de type ClasseY .
```

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

### 2.6. Caractéristiques des classes enveloppes ("wrapper class") :

- Les instances des classes **Integer, Long, Float, Double, Boolean** (et également **String**) sont **immuables (immutable)**. Cela signifie que pour changer la valeur, on ne peut pas modifier l'instance courante et qu'il faut reconstruire un nouvel objet qui remplacera l'ancien .

```
Integer objI = new Integer(5) ;  
objI.setValue(6) ; //n'existe pas !  
objI = new Integer(6) ; //nouvel objet (avec nouvelle valeur) remplaçant l'ancien.
```

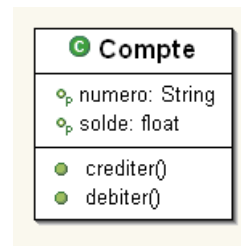
- Une variable de type int, double, ... ne peut jamais avoir la valeur null (0 mais pas null)
- Une variable de type Integer, Double , ... peut avoir la valeur null** (quelquefois par défaut si attribut d'une classe).  
La valeur **null** signifie quelquefois "*pas saisie*" dans interface graphique .  
La valeur **null** signifie quelquefois "*inconnue*" dans colonne d'une table (null au sens SQL) .

### 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 ici 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:

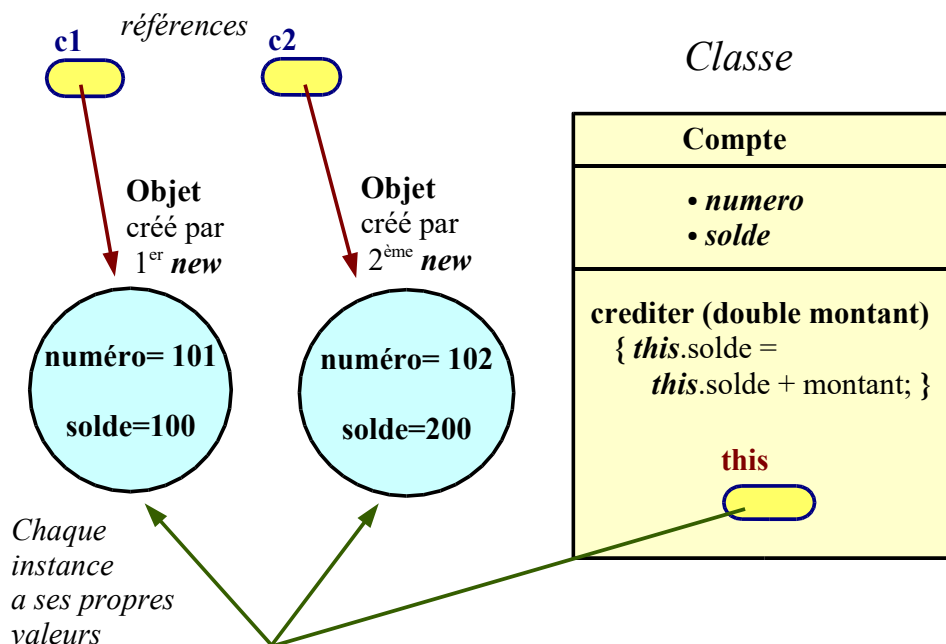
```
public class MyApp {

    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 a 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);
```

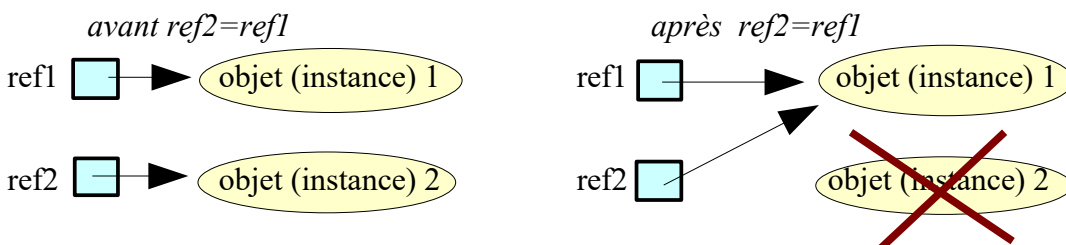
Appel classique de méthode (fonction interne) sur un objet Java:

```
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'affectation **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** `getXxx()` et `setXxx(...)` permettant respectivement de *recupérer* 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**" (quelquefois appelé **POJO** : Plain Old Java Object ou Plain Ordinary Java Object) .

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; }

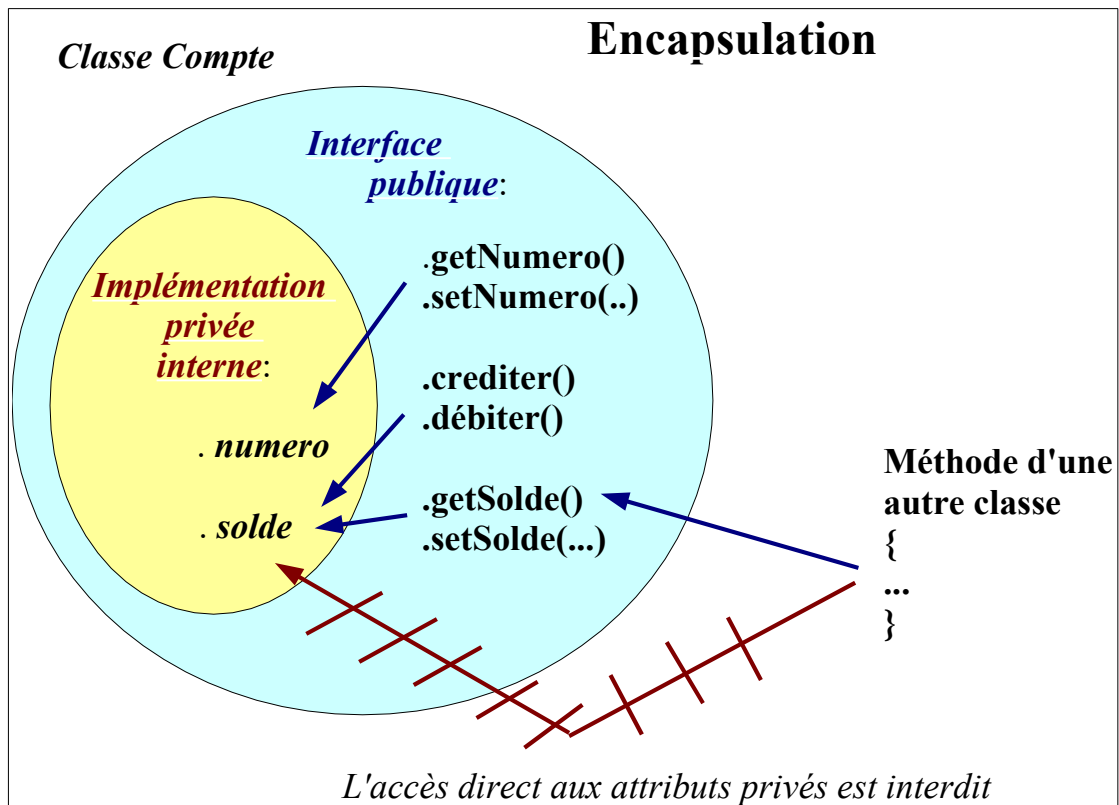
    public float getSolde() { return solde; }
    public void setSolde(float solde) { this.solde = solde; }

// 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 **getXxx()** et **setXxx()** 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 Rectangle):

```
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.

**DTO , JavaBean , POJO et Record**

Dans beaucoup d'applications java , on a souvent besoin de manipuler deux types d'objets de données :

- les **entités persistantes** (avec valeurs stockées et récupérées en base de données)
- les **DTO (Data Transfert Object)** alias VO (Value Object) pour communication entre couches logicielles ou via le réseau (avec conversions Java <—> JSON ou XML)

**NB :**

- Des entités ou **DTOs** sont toujours programmables comme des JavaBeans/POJO .
- On éventuellement s'appuyer sur l'extension lombok (@Getter @Setter ...) pour simplifier le code des JavaBeans/POJO.
- Depuis java 16,17 un nouveau mot clef **record** permet de coder de manière très compacte des classes de données spéciales qui sont complètement immuables/immutables .

**Ces "record"s , présentés en détails dans l'annexe "nouveauautés des versions 12-17"**  
sont maintenant utilisables pour mettre en œuvre des **DTOs**

## 4. Ramasse miettes (G.C.)

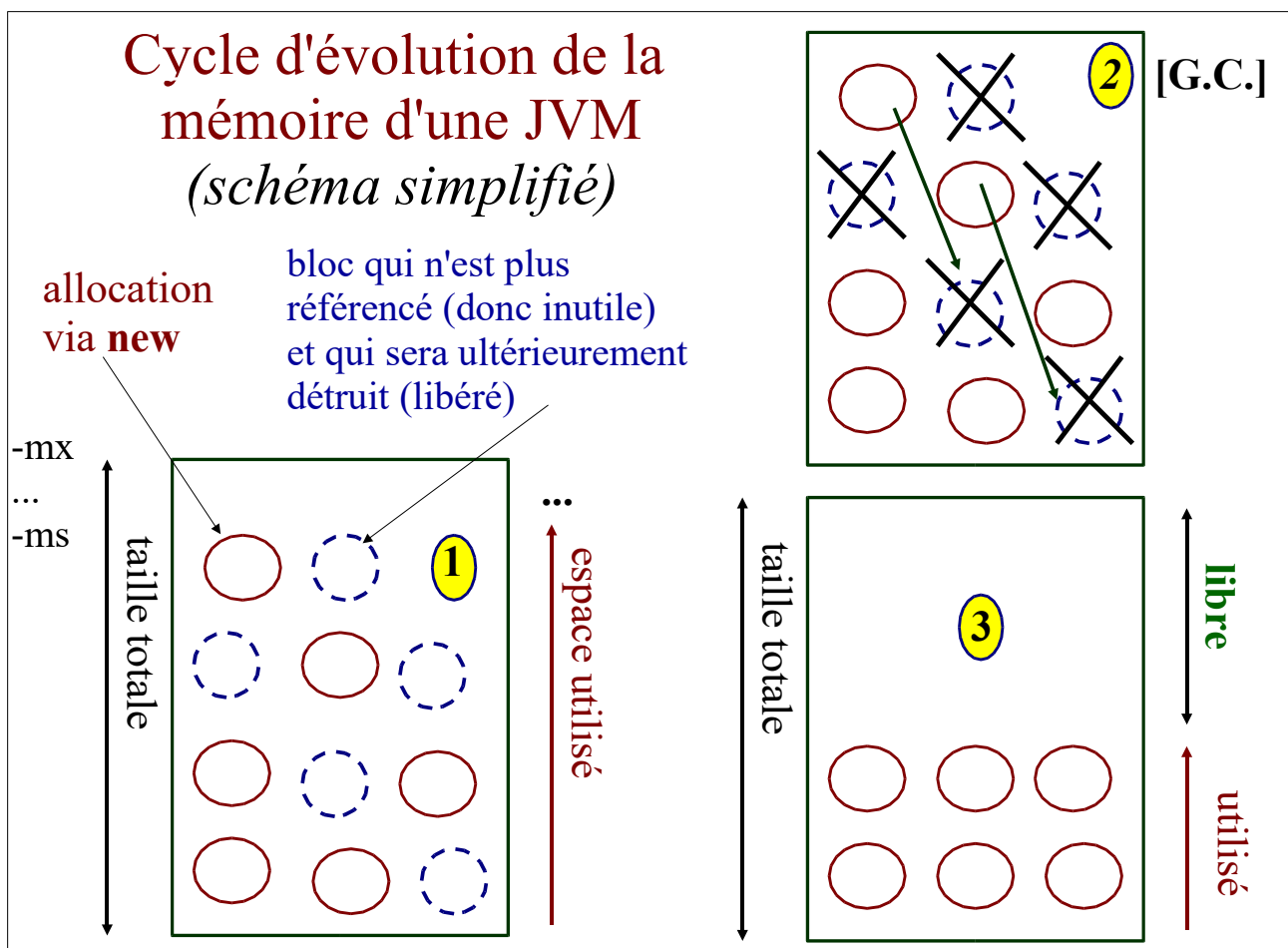
### 4.1. Destruction automatique des objets

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:

<i>opérateurs</i>	<i>Utilisations</i>	<i>Sémantiques</i>
+	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 ( $1 \wedge 1 ==> 0$ )
&	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++	incrémementation (pré/post)
--	--expr / expr--	décrémementation (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...0000001** 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<sup>3</sup> = 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	<b>&amp;</b> bit à bit ou booléen	gauche-->droite
8	<b>^</b> bit à bit ou booléen	gauche-->droite
9	<b> </b> bit à bit ou booléen	gauche-->droite
10	<b>&amp;&amp;</b> booléen	gauche-->droite
11	<b>  </b> booléen	gauche-->droite
12	<b>?:</b> (Expression conditionnelle si?alors:sinon )	droite-->gauche
13	<b>=   *=   /=   +=   -=   %=</b> <b>&lt;&lt;=   &gt;&gt;=   &amp;=   ^=   &gt;&gt;&gt;=</b>	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<Object> liste = new java.util.ArrayList<>();
liste.add(new String("abc"));
liste.add(new Integer(5));
```

...

```
java.util.Iterator<Object> 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");
}
```

NB: l'instruction switch/case a été améliorée dans les versions récentes de java. L'annexe sur les nouveautés de java 12 à 17 sera intéressante à étudier ultérieurement pour approfondir le sujet .

### 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 pour continuer */ ; 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, StringBuilder

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

Types	caractéristiques	valeurs littérales
<b>char</b>	caractère <b>UNICODE</b> (codé sur 2 octets)	'A' , '\u0041'
<b>String</b>	<b>Chaîne de caractères</b> (instance représentant la valeur globale de toute la chaîne).	"abc"
<b>StringBuilder</b> <b>StringBuffer</b>	<b>Buffer de caractères</b> servant à construire efficacement une chaîne via de multiples <b>concaténations</b> .	à convertir en <i>String</i>

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 :**

<b>.charAt(i)</b> avec i entre 0 et n-1	retourne le i <sup>ème</sup> caractère de la chaîne
<b>.length()</b>	retourne la longueur de la chaîne
<b>.equals(autreChaine)</b>	test si la chaîne courante à la même valeur qu'une autre
<b>.substring(firstPos,lastPos+1);</b>	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 (si plusieurs threads) !!!
StringBuilder buffer = new StringBuilder(64); // très souvent encore mieux !!!
for(int i=0;i<64;i++)
    buffer.append("");
String ch = buffer.toString();
```

**Quelques autres méthodes disponibles sur la classe String:**

<b>indexOf</b> (String chMotif ...)	retourne la première position où la sous-chaîne est trouvée
<b>toLowerCase</b> ()	retourne une nouvelle instance en <i>minuscules</i>
<b>toUpperCase</b> ()	retourne une nouvelle instance en <i>MAJUSCULES</i>
<b>String.valueOf</b> (...)	convertit une valeur de type <i>int</i> , <i>double</i> , ... en <i>String</i>
<b>split</b> (String separateur)	construit un tableau des sous chaines séparées par un séparateur

```
String s = "un;deux;trois" ;
String[] parties = s.split(";"); //parties[0] vaut "un", parties[1] vaut "deux" et parties[2] vaut "trois"
for(int i=0 ; i < parties.length ; i++) System.out.println(parties[i]) ;
```

## 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_regex="[0-9]*";
if(ligne.matches(grep_regex))
    ....
```

```
String newLigne = ligne.replaceAll(replace_regex,replace_text);
```

## 7.3. TextBlocs

NB : depuis java 15,16,17 , on peut englober via des *triples doubles quotes* le longues chaînes de caractères (réparties sur plusieurs lignes , avec à l'intérieur d'éventuels caractères " ) appelées *TextBlocs* :

```
String myJsonString =
""""
{
"taille" : 166 ,
"nom" : "toto"
}
"""" ;
```

## 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<sup>ème</sup>** élément.

*Syntaxe:*

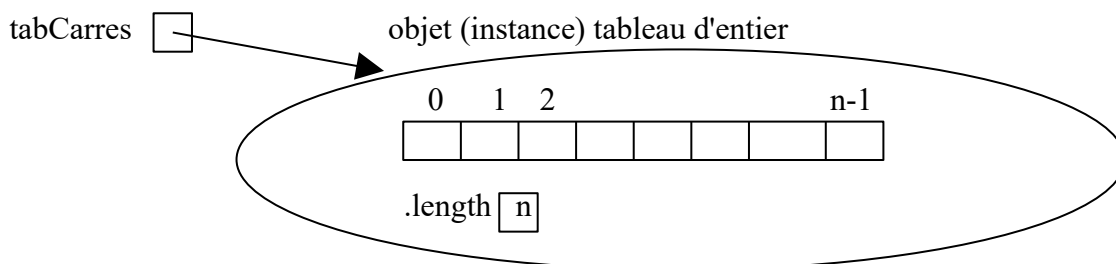
```
TypeElt [ ] refSurTableau=null; // ou bien TypeElt refSurTableau[]=null;
...
refSurTableau = new TypeElt[tailleDuTableau];
...
refSurTableau[positionElt] = valeur;
```

*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:

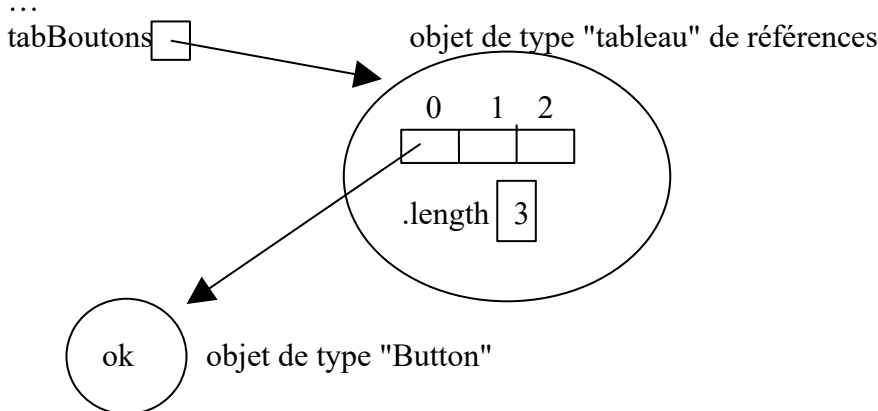
```
int[] tabCarres = new int[20]; // déclaration et construction possible sur la même ligne
for(int i= 0; i<tabCarres.length; i++)
    tabCarres[i] = i * i;
```



```
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:

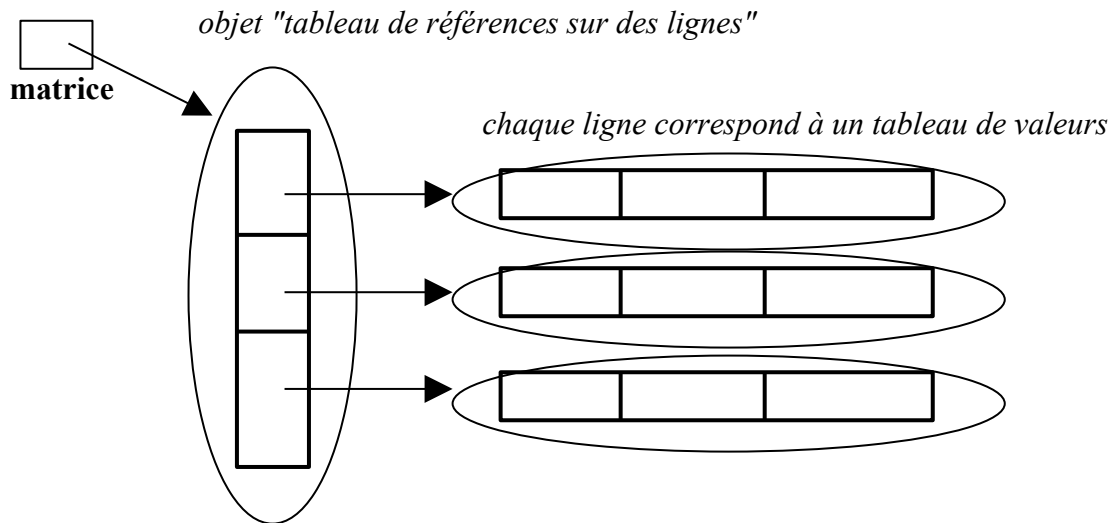
Toute tentative d'accès à un élément dont l'index est hors de l'intervalle  $[0, \text{tableau.length}-1]$  provoquera la levée d'une exception de type *ArrayIndexOutOfBoundsException*.

**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 **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**.  
La classe **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" } }
```



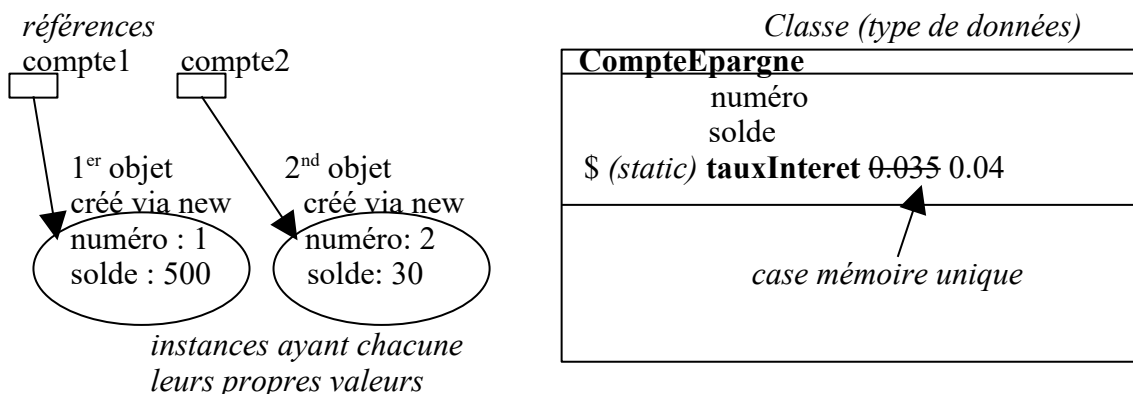
## 9. Méthode et variables de classes

### 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.setTauxInteret(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

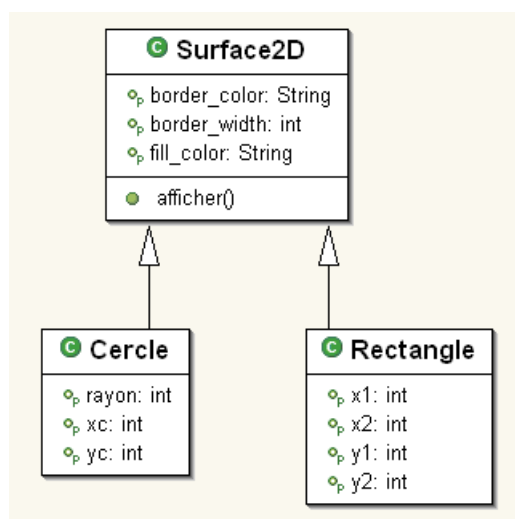
Lorsque l'on souhaite définir une **nouvelle classe dont une partie est déjà implémentée par une classe de base existante** et qui d'autre part doit comporter de **nouvelles spécificités**, il suffit alors de la faire **hériter** de la classe de base.

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. Depuis les versions 16,17 de java , le nouveau mot clef "**sealed**" (signifiant *scellé*) permet d'indiquer que seulement certaines classes auront le droit d'hériter d'une classe bien particulière (via le mot clef *permits*).  
*L'annexe sur les nouveautés de java 12à17 permet d'approfondir si besoin le le sujet.*
- 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(); // instantiation 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, ...

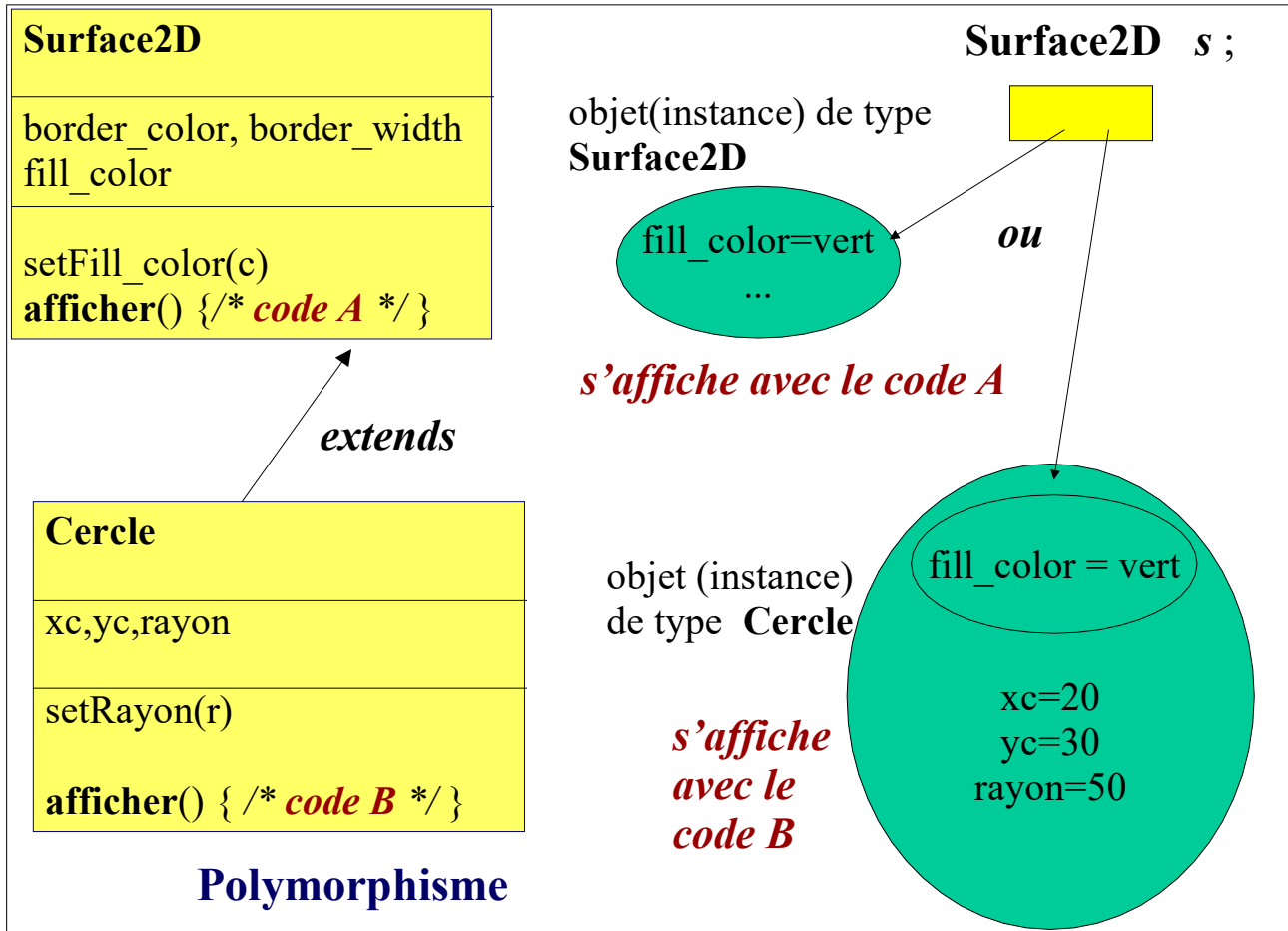
## 1.5. Redéfinition de méthode & opérations abstraites

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** défini 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** défini 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.

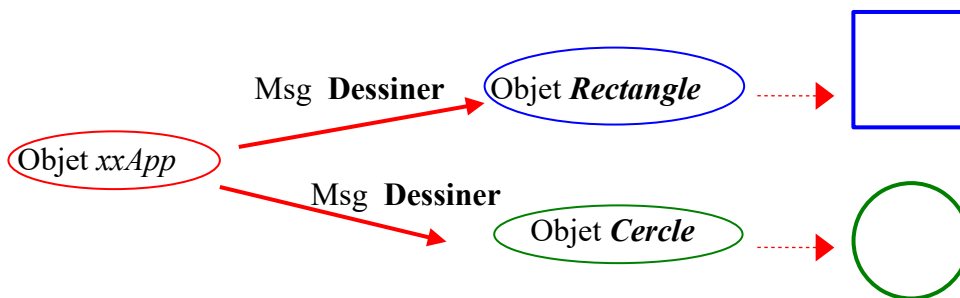
## 2.1. Liaison dynamique (formalisation du polymorphisme)

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 indispensables 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); }
}
```

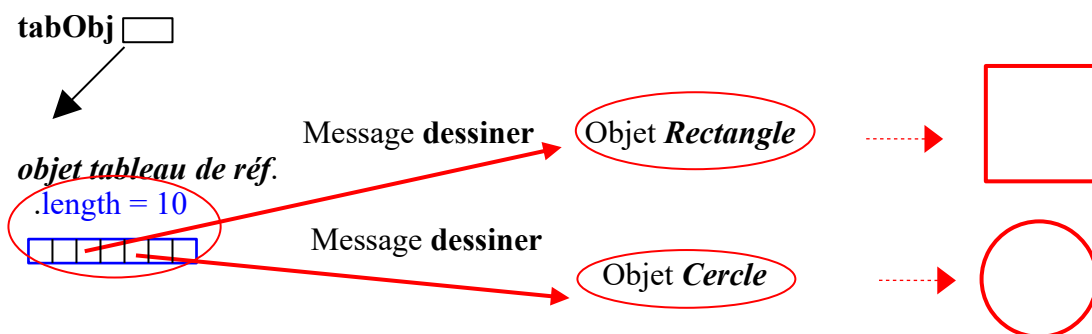
```
ObjetGraphique obj; // variable générique pouvant référencer tout type d'objet graphique
obj= new ObjetGraphique(); // new direct impossible car la classe est abstraite.
```

```

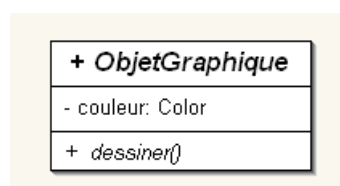
...
Cercle c1 = new Cercle(152.3 /*xc*/,15.66 /*yc*/,465.2 /*rayon*/);
Rectangle r1 = new Rectangle( 12.6 /*x1*/, 45.3 /*y1*/, 17.9 /*x2*/, 198.12 /*y2*/);
...
obj = r1;
obj.dessiner(g); // dessine un rectangle (polymorphisme)
obj = c1;
obj.dessiner(g); // dessine un cercle (polymorphisme)

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:

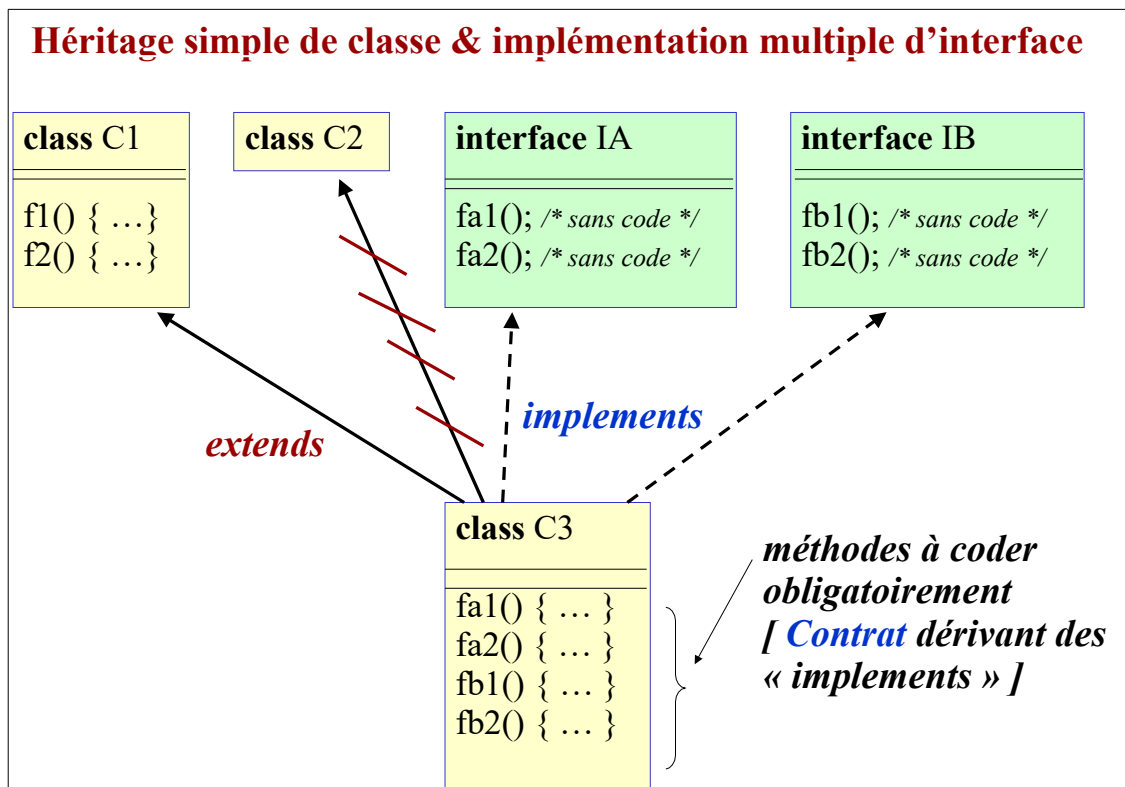


## 4. Interfaces

### 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:

- Rétérencer 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
```

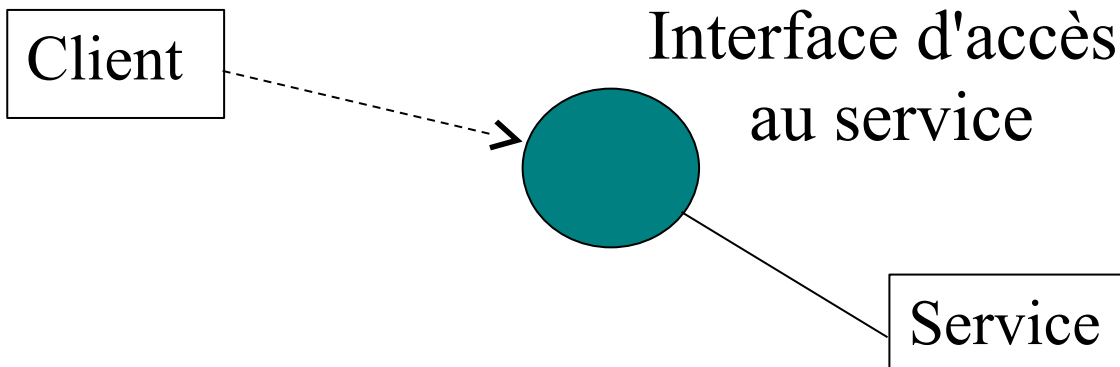
#### Héritage entre interfaces:

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 et pourra hériter de n'importe quelle 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 - Éléments 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 **pom.xml** une configuration **maven** (située à la racine d'un projet) et permettant de construire automatiquement une archive "**myAppli.jar**" en fonction de tout le contenu du projet :

##### **pom.xml**

```
<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>
  <groupId>org.mycontrib.xyz</groupId>
  <artifactId>myAppli</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
```



```

<properties>...</properties>
<dependencies>...</dependencies>
<build>
  <finalName>${project.artifactId}</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <configuration>
        <archive>
          <manifest>
            <mainClass>org.mycontrib.xyz.App</mainClass>
          </manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>

```

**mvn package** permet de générer une archive "**myAppli.jar**" comportant le fichier *Manifest* suivant:

#### META-INF/Manifest.mf

```

Manifest-Version: 1.0
Main-class:    org.mycontrib.xyz.App

```

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

NB: Le démarrage de l'application java via un double click ne fonctionne que dans le contexte suivant :

- ordinateur windows
- java installé et configuré dans le PATH
- le programme java lancé ne s'arrête pas aussitôt (il comporte une fenêtre graphique pour que l'on puisse le voir , ex : *JOptionPane.showMessageDialog(null,"Bienvenue MyApp");* )

Pour un lancement/démarrage plus universel (sans les contraintes précédentes) , on aura besoin d'écrire un script de ce genre :

```

set JAVA_HOME=C:\Program Files\java\jdk17
REM set JAVA_HOME=C:\Program Files\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.2.v20220201-1208\jre
set PATH=%JAVA_HOME%\bin
cd "%~dp0"
cd target
java -jar myAppli.jar
pause

```

## 1.3. javadoc

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

```
/**
 * @author didier
 *
 * Cette classe sert a ...
 */
....
```

**mvn javadoc:javadoc** (ou une ligne de commande équivalente sans maven) permet de **générer automatiquement une documentation HTML** sur les classes du projet .  
Le point d'entrée de cette documentation est **target/site/apidocs/index.html** .

OVERVIEW	PACKAGE	CLASS	USE	TREE	INDEX	HELP																					
SUMMARY: NESTED   FIELD   CONSTR   METHOD    DETAIL: FIELD   CONSTR   METHOD																											
Package tp <b>Class Bagage</b> java.lang.Object <sup>↳</sup> tp.Bagage All Implemented Interfaces: Transportable <pre>public class Bagage extends Object<sup>↳</sup> implements Transportable</pre> Author: d2fde Cette classe sert à transporter des affaires																											
<b>Field Summary</b> <table border="1"> <thead> <tr> <th>Fields</th> </tr> <tr> <th>Modifier and Type</th> <th>Field</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>static final Double<sup>↳</sup></td> <td>POIDS_MOYEN</td> <td></td> </tr> <tr> <td>static final Double<sup>↳</sup></td> <td>VOLUME_MOYEN</td> <td></td> </tr> </tbody> </table>							Fields	Modifier and Type	Field	Description	static final Double <sup>↳</sup>	POIDS_MOYEN		static final Double <sup>↳</sup>	VOLUME_MOYEN												
Fields																											
Modifier and Type	Field	Description																									
static final Double <sup>↳</sup>	POIDS_MOYEN																										
static final Double <sup>↳</sup>	VOLUME_MOYEN																										
<b>Constructor Summary</b> <table border="1"> <thead> <tr> <th>Constructors</th> </tr> <tr> <th>Constructor</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>Bagage()</td> <td></td> </tr> <tr> <td>Bagage(Double<sup>↳</sup> poids, Double<sup>↳</sup> volume)</td> <td></td> </tr> </tbody> </table>							Constructors	Constructor	Description	Bagage()		Bagage(Double <sup>↳</sup> poids, Double <sup>↳</sup> volume)															
Constructors																											
Constructor	Description																										
Bagage()																											
Bagage(Double <sup>↳</sup> poids, Double <sup>↳</sup> volume)																											
<b>Method Summary</b> <table border="1"> <thead> <tr> <th>All Methods</th> <th>Instance Methods</th> <th>Concrete Methods</th> </tr> <tr> <th>Modifier and Type</th> <th>Method</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>double</td> <td>getPoids()</td> <td></td> </tr> <tr> <td>double</td> <td>getVolume()</td> <td></td> </tr> <tr> <td>void</td> <td>setPoids(Double<sup>↳</sup> poids)</td> <td></td> </tr> <tr> <td>void</td> <td>setVolume(Double<sup>↳</sup> volume)</td> <td></td> </tr> <tr> <td>String<sup>↳</sup></td> <td>toString()</td> <td></td> </tr> </tbody> </table>							All Methods	Instance Methods	Concrete Methods	Modifier and Type	Method	Description	double	getPoids()		double	getVolume()		void	setPoids(Double <sup>↳</sup> poids)		void	setVolume(Double <sup>↳</sup> volume)		String <sup>↳</sup>	toString()	
All Methods	Instance Methods	Concrete Methods																									
Modifier and Type	Method	Description																									
double	getPoids()																										
double	getVolume()																										
void	setPoids(Double <sup>↳</sup> poids)																										
void	setVolume(Double <sup>↳</sup> volume)																										
String <sup>↳</sup>	toString()																										

**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:

<https://docs.oracle.com/en/java/javase/17/docs/api/>

All Modules	Java SE	JDK	Other Modules
Module	Description		
java.base	Defines the foundational APIs of the Java SE Platform.		
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs.		

  
**Packages**

Exports	
Package	Description
java.io	Provides for system input and output through data streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java programming language.

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

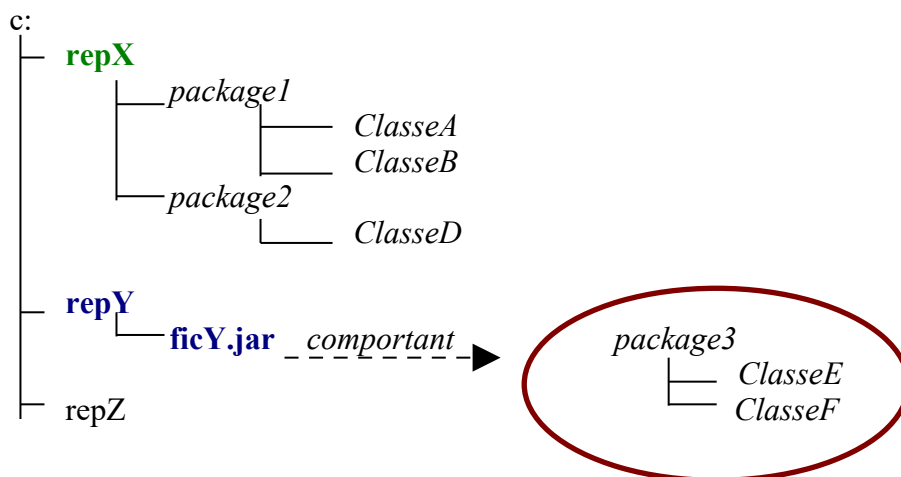
<b>Module</b> java.base <b>Package</b> java.text <b>Class</b> SimpleDateFormat java.lang.Object java.text.Format java.text.DateFormat java.text.SimpleDateFormat <b>All Implemented Interfaces:</b> Serializable, Cloneable	<b>Constructor Summary</b> <b>Constructors</b> Constructor SimpleDateFormat()  SimpleDateFormat(String pattern)  SimpleDateFormat(String pattern, DateFormatSymbols formatSymbols)  SimpleDateFormat(String pattern, Locale locale)
---	--

## 1.4. 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 certaines archives que la machine virtuelle consulte systématiquement :

- ... \jre\lib\rt.jar (pour java 1 à 8)
- ./jmods/java.base.jmod , java.xyz.jmod (à partir de java 9)

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

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

## 1.5. 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.

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

## 1.7. Résumé sur les modificateurs d'accès:

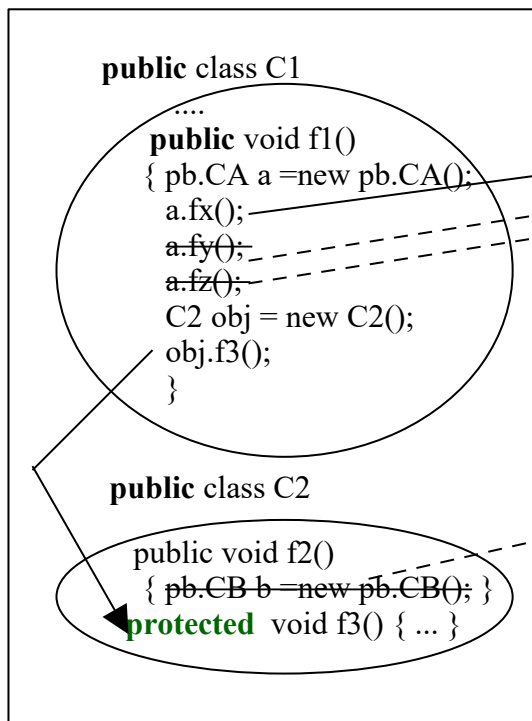
Accès depuis le package courant:

modificateur d'accès (m)	héritage possible d'une classe déclarée <i>m</i> pour une sous-classe du même package	membre déclaré <i>m</i> 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

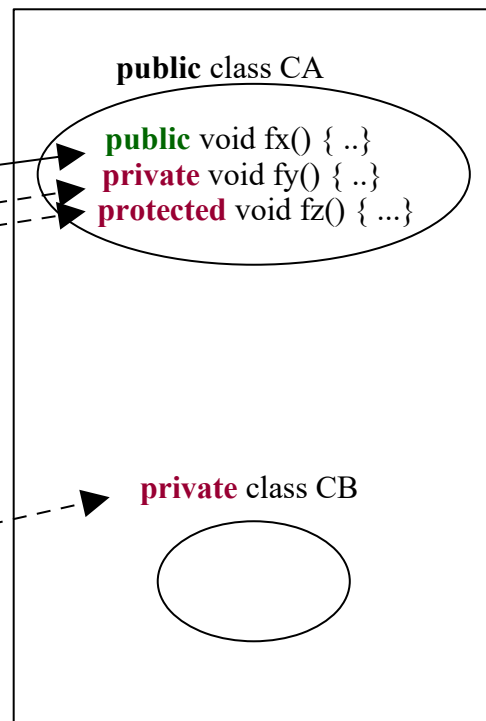
Accès depuis un autre package:

modificateur d'accès (m)	accès (ou héritage) possible vers une classe déclarée <i>m</i> pour une classe d'un autre package	membre déclaré <i>m</i> 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

package pa



package pb



## 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**.
- ♦ **Capter 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 lancée, 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:

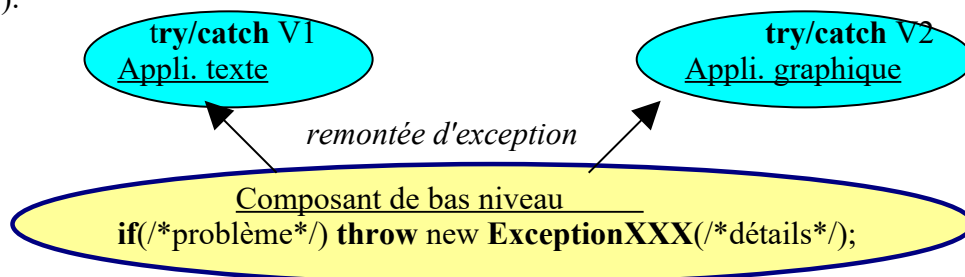
- Afficher un message d'erreur en mode texte (sur la console).
- Afficher un message en mode graphique dans une boîte de dialogue.
- Retourner l'erreur vers le client si l'erreur s'est produite coté serveur.
- Enregistrer le problème dans un fichier le log.
- ...

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.

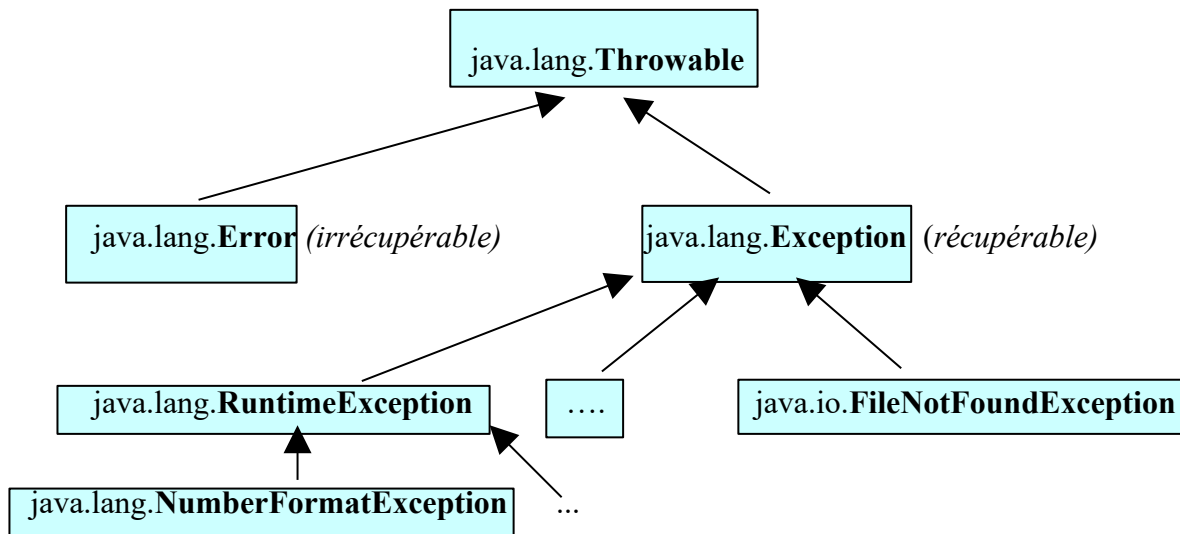
L'autre solution qui consiste à retourner systématiquement un code d'erreur (éventuellement du genre ERROR\_SUCCESS) n'est pas non plus très élégant car il oblige le code appelant à systématiquement tester (à coup de if/else ou switch) la valeur de retour de la fonction de bas niveau pour savoir si tout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

Le mécanisme de traitement des exceptions apporte la meilleure 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'un 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 , on parle alors en terme de "*unchecked exception*" )

## 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(NullPointerException e1)
{
... // Gestion de l'exception e1 qui est une instance de NullPointerException
... // ou d'une sous classe.
}
catch(NullPointerException 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(NullPointerException e1, NullPointerException e2){ ..... }
```

## 2.5. Niveau intermédiaire – alternatives de traitement (exceptions)

```
... class MyApp {
...main(...)
{ Cxxx objX ...
  try{ objX.f1()}
  catch(Exception ex)
  {
    ex.printStackTrace();
  }
}
```

Remontée  
d'exception

```
... class Cyyy {
...f1a(...) throws MyException
{...
  if(...)
    throw new MyException(...);
  ...
}
```

```
... class Cxxx {
...f1(...)
{ Cyyy objY ...
  try{ objY.f1a()}
  catch(MyException myex)
  {
    myex.printStackTrace();
    ... }
}

OU BIEN

... f1(...) throws MyException
{
  Cyyy objY ...
  objY.f1a();
}

OU BIEN ...

... catch(MyException myex)
{ myex.printStackTrace();
  throw new OtherException(...);}
}
```

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 :

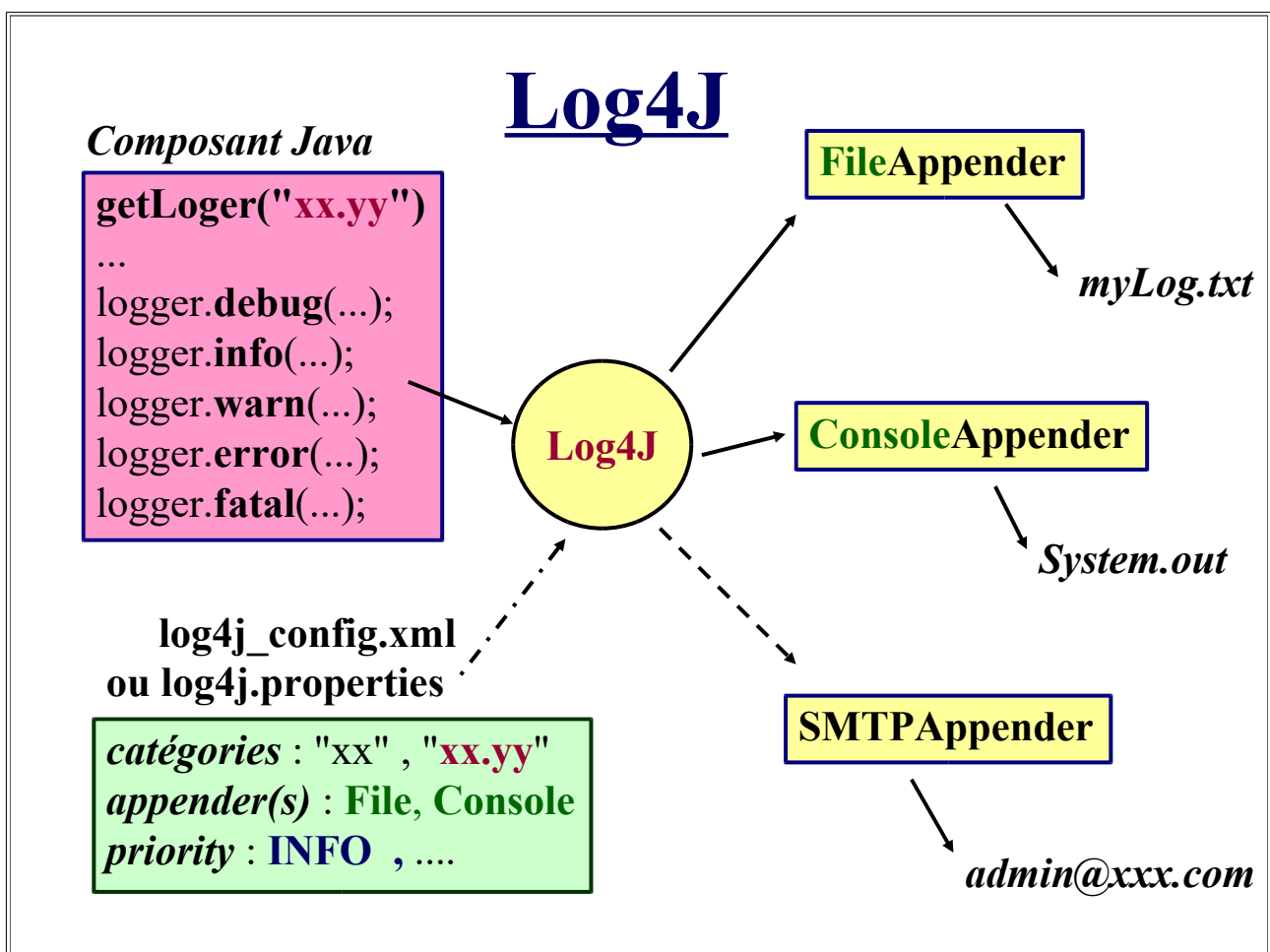
- générer des logs sur le problème vu localement (avec détails techniques)
- remonter une exception dans le format attendu par le niveau appelant

### 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 façade for java) qui (comme commons-logging) est également une petite API de haut niveau déléguant 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 paramètres.

Will Output Messages Of Level		DEBUG	INFO	WARN	ERROR	FATAL
Logger Level	DEBUG					
	INFO					
	WARN					
	ERROR					
	FATAL					
	ALL					
	OFF					

### 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* (ou des équivalents) 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 log4j2)

*src/main/resources/log4j2.properties*

```
# Extra logging related to initialization of Log4j
# Set to debug or trace if log4j initialization is failing
status = warn
# Name of the configuration
name = ConsoleLogConfigDemo

# Console appender configuration
appender.console.type = Console
appender.console.name = consoleLogger
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = %d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

# RollingFileAppender name, pattern, path and rollover policy
appender.rolling.type = RollingFile
appender.rolling.name = fileLogger
appender.rolling.fileName= mylogs.log
appender.rolling.filePattern= mylogs_%d{yyyyMMdd}.log.gz
appender.rolling.layout.type = PatternLayout
appender.rolling.layout.pattern = %d{yyyy-MM-dd HH:mm:ss.SSS} %level [%t] [%l] - %msg%n
appender.rolling.policies.type = Policies

# Root logger level
#rootLogger.level = debug
rootLogger.level = warn
# Root logger referring to console appender
rootLogger.appenderRef.stdout.ref = consoleLogger

logger.xyz.name = tp.xyz
logger.xyz.level = debug
logger.xyz.appenderRef.stdout.ref = consoleLogger
logger.xyz.additivity = false
```

## 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:

<b>user.home</b>	home directory
<b>java.home</b>	répertoire d'installation de java
<b>java.class.path</b>	path pour trouver et charger les fichiers .class
<b>os.name</b>	nom du système d'exploitation hôte
<b>line.separator</b>	ex: "\n" sous Unix
<b>path.separator</b>	Ex: ":" sous Unix , ";" sous PC
<b>user.name</b>	nom de l'utilisateur courant
<b>user.dir</b>	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 JAVA_HOME=C:\Program Files\java\jdk17
set PATH=%JAVA_HOMEPATH%
set CLASSPATH=%CLASSPATH%;c:\rep\xxx.jar;c:\rep\
java -classpath c:\rep\autreApi.jar -Dprop1=val1 -Dprop2=val2 package_p.MonAppli arg1
arg2 .... argn    > ficRes.txt    2> ficErr.txt
```

## 5. 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 ClassImbrique {
        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 ici 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() {   ClassImbrique obj = new ClassImbrique();
                           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 ici 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.

## 6. Collections (depuis le jdk 1.2)

<div> <div>Implémentations</div> <div>→</div> </div> <div> <div>Interfaces</div> <div>↓</div> </div>	Hash Table	Resizable Array	Balanced Tree	Linked List	Vielles 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 <b>add</b> (Object o)	retourne false si l'élément y était déjà ou si ...
boolean <b>addAll</b> (Collection c)	
void <b>clear</b> ()	vide tout
boolean <b>contains</b> (Object o)	objet contenu dans la collection ?
boolean <b>isEmpty</b> ()	
Iterator <b>iterator</b> ()	retourne un itérateur pour parcourir la collection
boolean <b>remove</b> (Object o)	
int <b>size</b> ()	Nombre d'éléments de la collection
Object[] <b>toArray</b> ()	
...	

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

```

Iterator<Object> 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:

<i>principales méthodes</i>	<i>détails</i>
Object <b>get</b> (int index)	retourne l'objet à la position indiquée
int <b>indexOf</b> (Object o)	retourne -1 si l'objet ne fait pas partie de la liste
int <b>lastIndexOf</b> (Object o)	idem pour dernière occurrence trouvée
ListIterator <b>listIterator</b> ()	
Object <b>set</b> (int index, Object o)	remplace l'élément en position index
List <b>subList</b> (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()** ).

Exemple :

```
List<String> liste = new ArrayList<>();
liste.add("janvier");
liste.add("fevrier");
liste.add("mars");

//parcours avec code proche d'un parcours de tableau ordinaire:
int n = liste.size();
System.out.println("n="+n);
for(int i=0;i<n;i++) {
    Object obj = liste.get(i);
    System.out.println("obj="+obj);
}
```

```
//parcours avec un itérateur (années 2000-2005):
//Iterator it = liste.iterator();
Iterator<String> it = liste.iterator();
while(it.hasNext()) {
    String valeur = /*(String)*/ it.next();
    System.out.println("valeur="+valeur);
}

//parcours moderne avec boucle for() au sens forEach():
for(String sVal : liste) {
    System.out.println("sVal="+sVal);
}
```

### 6.3. Interface Map

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

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

Exemple :

```
Map<Integer,String> mapMois = new HashMap<>();
mapMois.put(1, "janvier");
//...
mapMois.put(12, "decembre");

String nomDuMoisNumero12 = mapMois.get(12);
System.out.println("nomDuMoisNumero12="+nomDuMoisNumero12);//affiche decembre
```

## 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);
}
```

```
List<Double> listeNumerique = new ArrayList<>();
```

```
listeNumerique.add(12.0);
```

```
listeNumerique.add(4.0);
```

```
listeNumerique.add(8.0);
```

```
Collections.sort(listeNumerique); //les Integer,Long,Deouble,String se comparent entre eux automatiquement
```

```
public class CompareurDePersonne implements java.util.Comparator<Personne>{
    @Override
    public int compare(Personne o1, Personne o2) {
        //return o1.getTaille() - o2.getTaille();
        if(o1.getNom() != null)
            return o1.getNom().compareTo(o2.getNom());
        else return -1;
    }
}
```

```
List<Personne> listePersonnes = new ArrayList<>();
```

```
listePersonnes.add(new Personne("jean","Bon","jb@xy.com",177));
```

```
listePersonnes.add(new Personne("axelle","Aire","aa@xy.com",167)); //...
```

```
CompareurDePersonne compareurDePersonne = new CompareurDePersonne();
```

```
Collections.sort(listePersonnes,compareurDePersonne);
```

```
for(Personne p : listePersonnes) {
```

```
    System.out.println("\t" + p); //"\t" pour tabulation , "\n" pour saut de ligne
```

```
}
```

## 7. Generics (depuis Java 5)

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

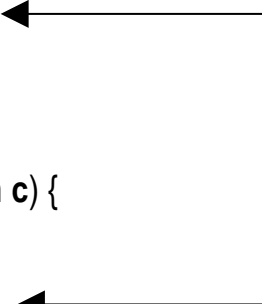
Soit une classe utilisant une Collection d'objet String et parcourue grâce à un itérateur :

```
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));
        printCollection(list);
    }

    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();
    }
}
```



*// incohérence  
// pas détectée  
// à la compilation*

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:

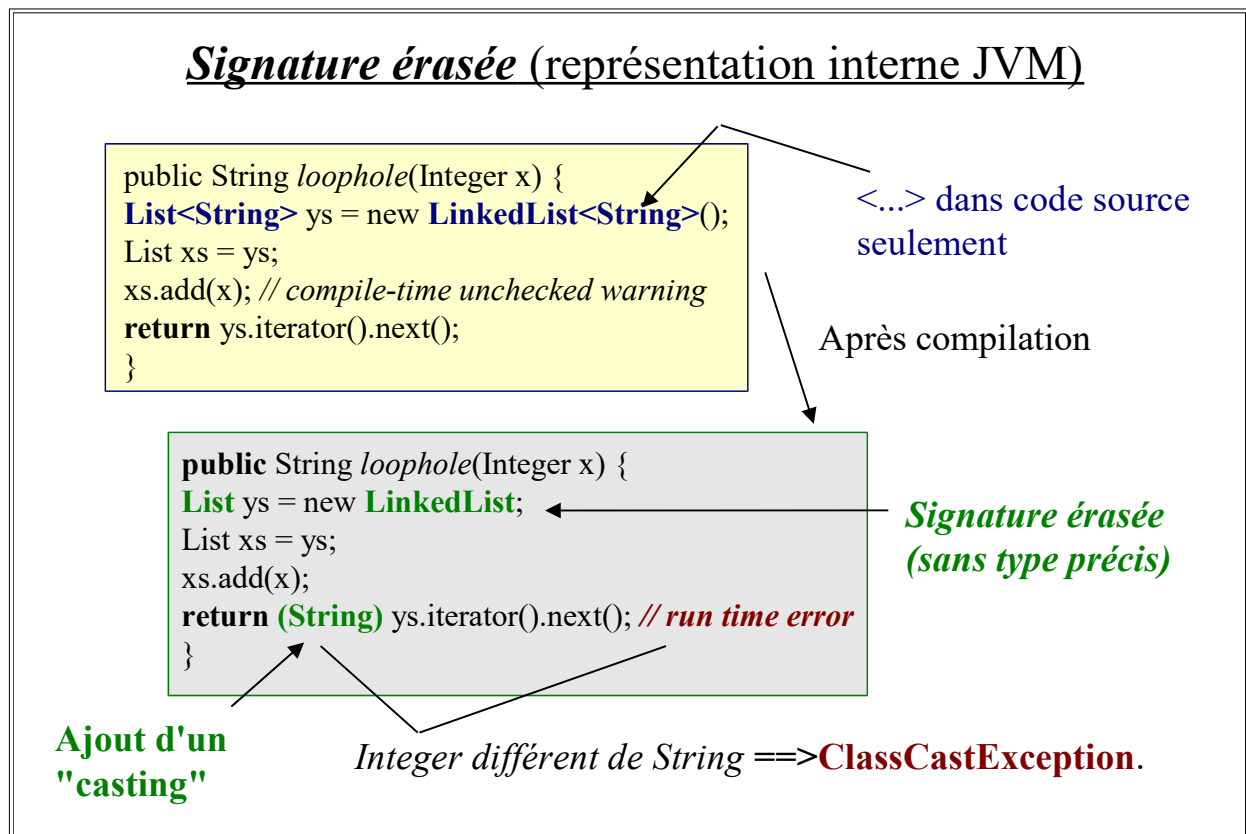
```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
    ...
}

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 nouvelles 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. Diamond operator (simplification depuis jdk 1.7)

```
MyStack<Integer> pileEntiers = new MyStack<>();
MyStack<String> pileChaines = new MyStack<>();
```

Le compilateur (devenu plus intelligent) est capable (par inférence) de déduire le type précis à instancier en se basant sur le type précis de l'interface .



## 7.5. 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-écrit avec ici 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-écrit :*

```
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.

## 8. Modules java 9+ (essentiel)

### 8.1. Problèmes et limitations en java 8

En Java 8 et antérieur, à l'exécution d'une application, la JVM recherche les classes utilisées par l'application dans :

- Les classes prédéfinies de la plate-forme Java : stockées dans l'**énorme** fichier **monolithique** `rt.jar` (*jusqu'à 53 Mo en Java 8 !*).
- Le **classpath** paramétré au lancement (un ensemble de chemins relatifs ou absolus vers des `.jars` ou des répertoires contenant des arborescences en package contenant des fichiers `“.class”` )

Selon d'éventuelles différences de contextes (ordinateurs différents, "jdk" différents, ...) , **le classpath n'est pas forcément le même à la compilation et à l'exécution** et cela peut engendrer `"java.lang.NoClassDefFoundError"` !!!!

En Java 8 et antérieur, à l'exécution d'une application, le **"classloader"** de la JVM a le comportement suivant :

- il charge les classes sur demande lors d'un `new` ou d'un appel à une méthode statique.
- il charge les classes linéairement (selon l'ordre des jars dans le classpath qui est donc très important) et s'arrête à la première classe correspondant au nom complet demandé.
- aucune vérification n'est faite à l'exécution sur l'existence de plusieurs occurrences d'une même classe.
- aucune vérification n'est faite au démarrage de l'application sur la présence de tous les jars / classes nécessaires au bon fonctionnement de l'application.
- le classpath est plat sans aucune notion de dépendances entre jar.

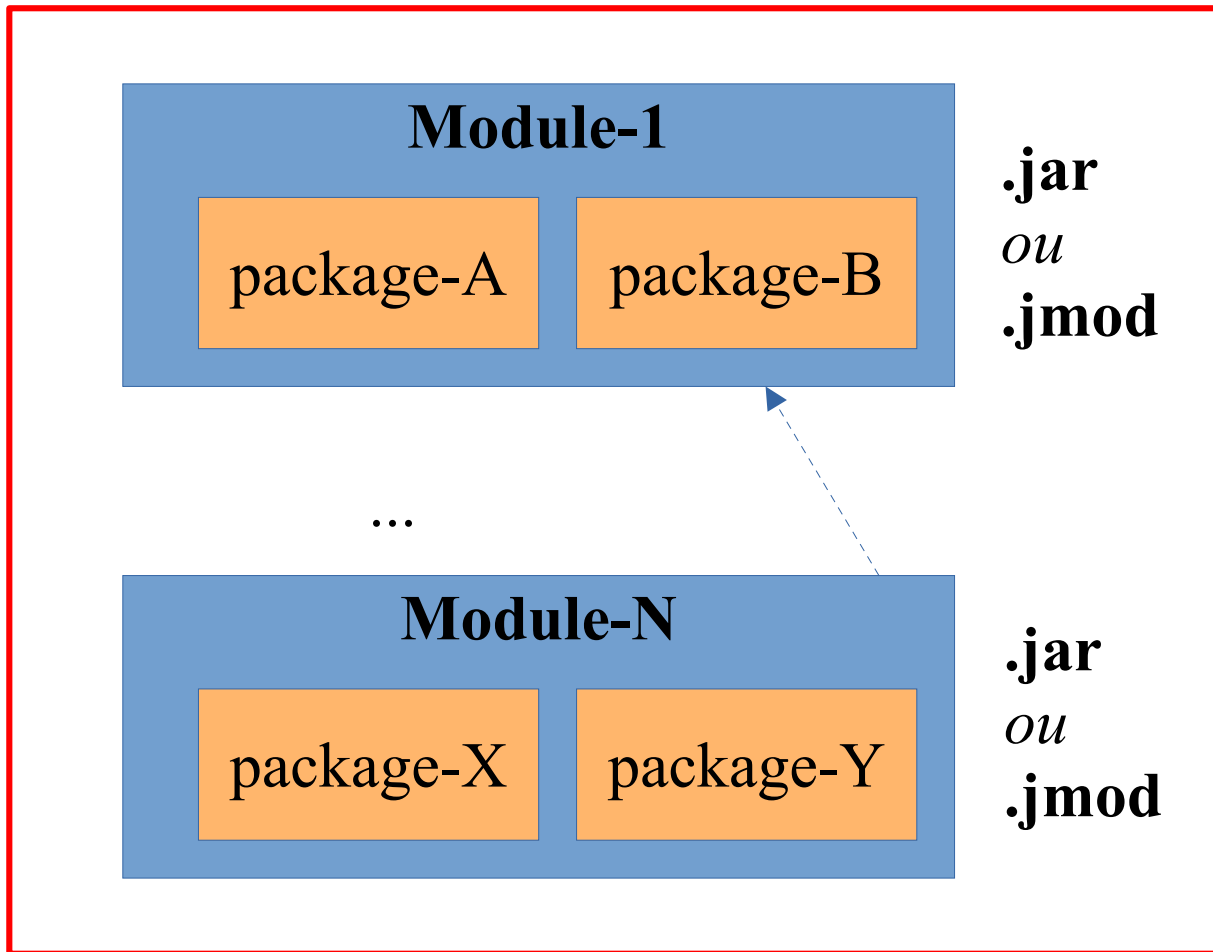
### 8.2. Projet JIGSAW (modularisation de la jvm depuis java >=9 )

Principaux **apports** de la **modularisation** de la JVM (depuis V9) :

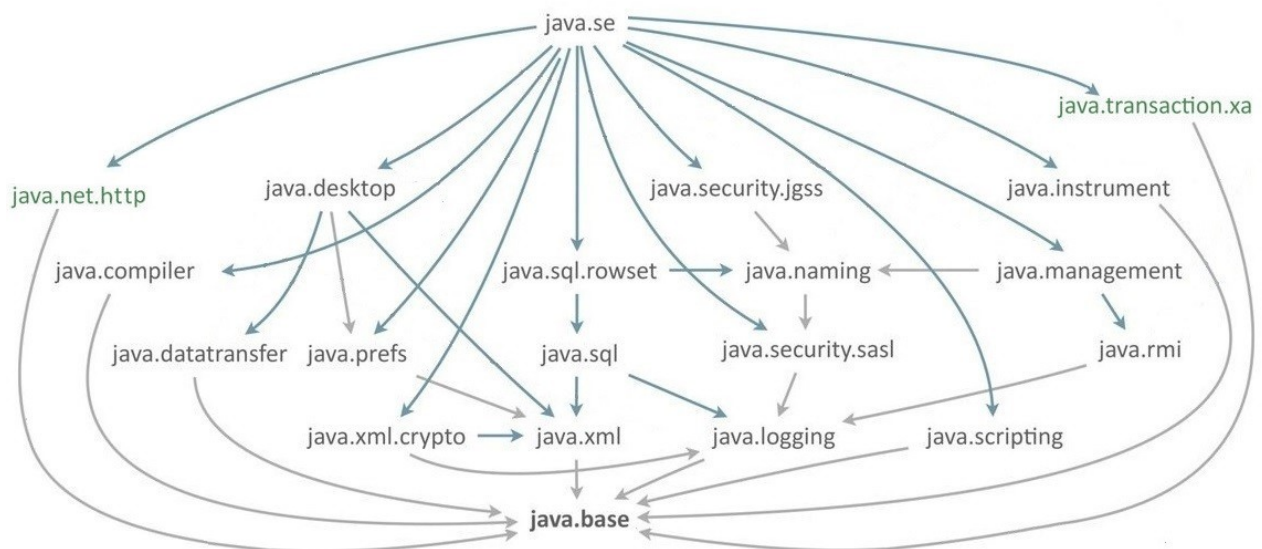
- **un classpath sous forme d'arbre de dépendances**
- **une vérification de la présence de tous les modules nécessaires à notre application au démarrage**, sans quoi l'application ne démarre pas.
- **un renforcement de la sécurité**, seuls les packages exportés explicitement par un module sont visibles par un autre
- la JVM elle-même est **modulaire** (*l'énorme fichier `rt.jar` n'existe plus depuis java 9*)  
(NB : la taille du JRE a sensiblement diminué à partir de java 11)

NB : Un module "java >=9" est un **.jar** comportant le fichier ***module-info.java*** (compilé en `module-info.class`) à la racine du module .

## Appli ou JVM modulaire depuis java 9










































































Exemple de graphe de dépendances entre modules d'un jdk récent :



**Modules du jdk 11 :**

Java &gt; jdk-11.0.4 &gt; jmods

Nom	Modifié le	Type	Taille
 java.base.jmod	19/08/2019 20:57	Fichier JMOD	18 021 Ko
 java.compiler.jmod	19/08/2019 20:57	Fichier JMOD	109 Ko
 java.datatransfer.jmod	19/08/2019 20:57	Fichier JMOD	50 Ko
 java.desktop.jmod	19/08/2019 20:57	Fichier JMOD	11 799 Ko
 java.instrument.jmod	19/08/2019 20:57	Fichier JMOD	116 Ko
 java.logging.jmod	19/08/2019 20:57	Fichier JMOD	110 Ko
 java.management.jmod	19/08/2019 20:57	Fichier JMOD	867 Ko
 java.management.rmi.jmod	19/08/2019 20:57	Fichier JMOD	87 Ko
 java.naming.jmod	19/08/2019 20:57	Fichier JMOD	436 Ko
 java.net.http.jmod	19/08/2019 20:57	Fichier JMOD	680 Ko
 java.prefs.jmod	19/08/2019 20:57	Fichier JMOD	53 Ko
 java.rmi.jmod	19/08/2019 20:57	Fichier JMOD	369 Ko
 java.scripting.jmod	19/08/2019 20:57	Fichier JMOD	44 Ko
 java.se.jmod	19/08/2019 20:57	Fichier JMOD	3 Ko
 java.security.jgss.jmod	19/08/2019 20:57	Fichier JMOD	638 Ko
 java.security.sasl.jmod	19/08/2019 20:57	Fichier JMOD	79 Ko
 java.smartcardio.jmod	19/08/2019 20:57	Fichier JMOD	57 Ko
 java.sql.jmod	19/08/2019 20:57	Fichier JMOD	74 Ko
 java.sql.rowset.jmod	19/08/2019 20:57	Fichier JMOD	184 Ko
 java.transaction.xa.jmod	19/08/2019 20:57	Fichier JMOD	4 Ko
 java.xml.crypto.jmod	19/08/2019 20:57	Fichier JMOD	635 Ko
 java.xml.jmod	19/08/2019 20:57	Fichier JMOD	4 380 Ko
 jdk.accessibility.jmod	19/08/2019 20:57	Fichier JMOD	484 Ko
 jdk.aot.jmod	19/08/2019 20:57	Fichier JMOD	274 Ko
 jdk.attach.jmod	19/08/2019 20:57	Fichier JMOD	38 Ko
 jdk.charsets.jmod	19/08/2019 20:57	Fichier JMOD	1 451 Ko
 jdk.compiler.jmod	19/08/2019 20:57	Fichier JMOD	6 904 Ko
⋮			
 jdk.crypto.cryptoki.jmod	19/08/2019 20:57	Fichier JMOD	305 Ko
 jdk.crypto.ec.jmod	19/08/2019 20:57	Fichier JMOD	148 Ko
 jdk.crypto.mscapi.jmod	19/08/2019 20:57	Fichier JMOD	61 Ko
 jdk.dynalink.jmod	19/08/2019 20:57	Fichier JMOD	159 Ko
 jdk.editpad.jmod	19/08/2019 20:57	Fichier JMOD	7 Ko
 jdk.hotspot.agent.jmod	19/08/2019 20:57	Fichier JMOD	2 298 Ko
 jdk.httpserver.jmod	19/08/2019 20:57	Fichier JMOD	98 Ko
 jdk.internal.ed.jmod	19/08/2019 20:57	Fichier JMOD	8 Ko
 jdk.internal.jvmstat.jmod	19/08/2019 20:57	Fichier JMOD	88 Ko
 jdk.internal.le.jmod	19/08/2019 20:57	Fichier JMOD	181 Ko
 jdk.internal.opt.jmod	19/08/2019 20:57	Fichier JMOD	80 Ko
 jdk.internal.vm.ci.jmod	19/08/2019 20:57	Fichier JMOD	399 Ko
 jdk.internal.vm.compiler.jmod	19/08/2019 20:57	Fichier JMOD	6 025 Ko
 jdk.internal.vm.compiler.manag...	19/08/2019 20:57	Fichier JMOD	12 Ko
 jdk.jartool.jmod	19/08/2019 20:57	Fichier JMOD	195 Ko
 jdk.javadoc.jmod	19/08/2019 20:57	Fichier JMOD	1 594 Ko
 jdk.jcmd.jmod	19/08/2019 20:57	Fichier JMOD	143 Ko
 jdk.jconsole.jmod	19/08/2019 20:57	Fichier JMOD	456 Ko
 jdk.jdeps.jmod	19/08/2019 20:57	Fichier JMOD	710 Ko
 jdk.jdi.jmod	19/08/2019 20:57	Fichier JMOD	841 Ko
 jdk.jdpw.agent.jmod	19/08/2019 20:57	Fichier JMOD	121 Ko
 jdk.jfr.jmod	19/08/2019 20:57	Fichier JMOD	414 Ko
 jdk.jlink.jmod	19/08/2019 20:57	Fichier JMOD	388 Ko
 jdk.jshell.jmod	19/08/2019 20:57	Fichier JMOD	626 Ko
 jdk.jsobject.jmod	19/08/2019 20:57	Fichier JMOD	6 Ko
 jdk.jstatd.jmod	19/08/2019 20:57	Fichier JMOD	33 Ko
⋮			
 jdk.localedata.jmod	19/08/2019 20:57	Fichier JMOD	9 327 Ko
 jdk.management.agent.jmod	19/08/2019 20:57	Fichier JMOD	80 Ko
 jdk.management.jfr.jmod	19/08/2019 20:57	Fichier JMOD	34 Ko
 jdk.management.jmod	19/08/2019 20:57	Fichier JMOD	68 Ko
 jdk.naming.dns.jmod	19/08/2019 20:57	Fichier JMOD	57 Ko
 jdk.naming.rmi.jmod	19/08/2019 20:57	Fichier JMOD	19 Ko
 jdk.net.jmod	19/08/2019 20:57	Fichier JMOD	21 Ko
 jdk.pack.jmod	19/08/2019 20:57	Fichier JMOD	131 Ko
 jdk.rmic.jmod	19/08/2019 20:57	Fichier JMOD	516 Ko
 jdk.scripting.nashorn.jmod	19/08/2019 20:57	Fichier JMOD	2 143 Ko
 jdk.scripting.nashorn.shell.jmod	19/08/2019 20:57	Fichier JMOD	56 Ko
 jdk.sctp.jmod	19/08/2019 20:57	Fichier JMOD	23 Ko
 jdk.security.auth.jmod	19/08/2019 20:57	Fichier JMOD	73 Ko
 jdk.security.jgss.jmod	19/08/2019 20:57	Fichier JMOD	24 Ko
 jdk.unsupported.desktop.jmod	19/08/2019 20:57	Fichier JMOD	14 Ko
 jdk.unsupported.jmod	19/08/2019 20:57	Fichier JMOD	18 Ko
 jdk.xml.dom.jmod	19/08/2019 20:57	Fichier JMOD	42 Ko
 jdk.zipfs.jmod	19/08/2019 20:57	Fichier JMOD	87 Ko

### 8.3. option "--list-modules" de java

*print\_list\_modules.bat*

```
set JAVA_HOME=D:\Prog\java\jdk-14.0.1_windows-x64_bin\jdk-14.0.1
"%JAVA_HOME%\bin\java" --list-modules
pause
```

--> affiche la liste des modules systèmes de la JRE (et du JDK) , soit par exemple :

```
java.base@14.0.1
java.compiler@14.0.1
java.datatransfer@14.0.1
java.desktop@14.0.1
...
jdk.xml.dom@14.0.1
jdk.zipfs@14.0.1
```

### 8.4. option "--module-path" de java

L'option **--module-path** d'une ligne de commande java permet de spécifier le "module path". C'est une **liste de un ou plusieurs répertoires qui comportent des modules** .

Exemple :

*prepare\_modules.bat*

```
cd /d "%~dp0"
REM cd ..
REM mvn package
REM cd /scripts
copy ..\mod_xx\target\mod_xx.jar .\my_modules
copy ..\mod_yy\target\mod_yy.jar .\my_modules
pause
```

*launch\_app\_withmodules.bat*

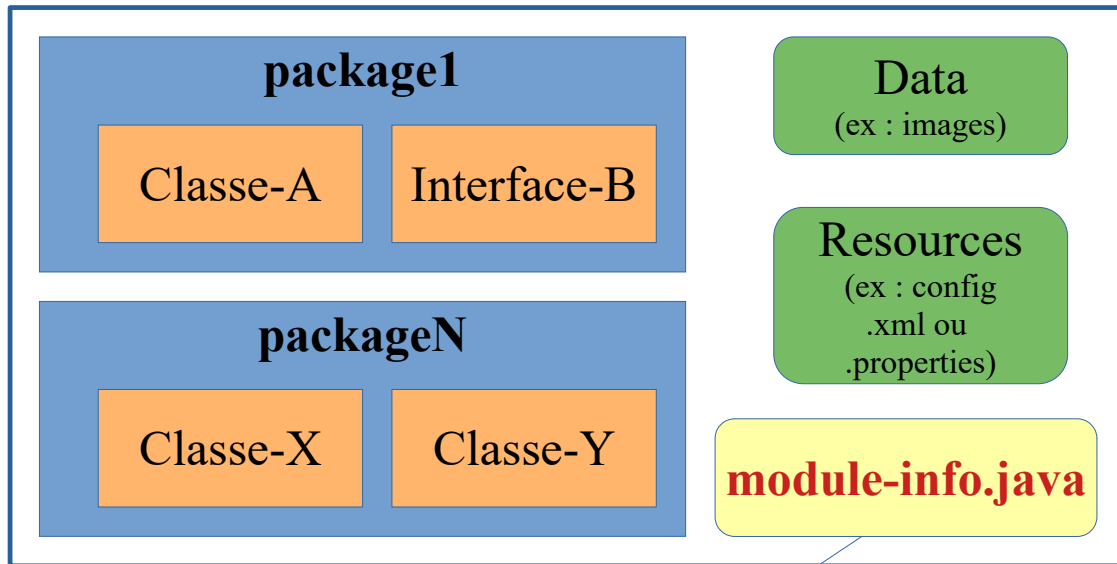
```
cd /d "%~dp0"
REM java --module-path modules_directory -m mainModuleName/mainClassFullName
java --module-path .\my_modules -m tp.module.modyy/tp.mod.mod_yy.app.MyApp
pause
```

## 8.5. Structure d'un module

Un module java (packagé en un ".jar" éventuellement renommé ) est **l'association** de :

- un **paquet de packages java** (un sous package précis "xxx.yyy.zzz" ne peut être rangé que dans un seul module )
- un **paquet d'éventuelles ressources** (fichiers annexes : configuration\_xml , images, ...)
- un fichier de configuration **module-info.java**

### Module java 9+



*NB: si module-info.java pas spécifié , alors module automatique (exports \* , requires \*)*

```
module m2 {
    exports package1;
    requires m1;
}
```

## 8.6. Types de modules

Types de modules	Caractéristiques
<b>System Modules</b>	modules prédéfinis livrés avec jre/jdk (listés via <b>java --list-modules</b> )
<b>Application Modules</b>	modules applicatifs ( <b>.jar</b> fabriqués <b>avec module-info.class</b> )
<b>Automatic Modules</b>	unofficial module ( <b>.jar</b> fabriqués <b>sans module-info.class</b> ) (with names based on ".jar" name) chargés depuis le --module-path . <i>Attention : sans module-info.class , ces modules ont par défaut un accès complet en lecture vers tous les autre modules chargés .</i>
<b>Unnamed Module</b>	" <b>module fourre tout sans nom</b> " comportant toutes les classes et packages chargés depuis des ".jar" via le --classpath (et pas le nouveau --module-path ) -> pour récupérer tout les éléments codés en java <= 8 .



# V - Prog. Fonctionnelle (lambda, stream, ...)

## 1. Java multi-paradigme "objet + fonctionnel"

### 1.1. Evolution importante dès java 8

Les premières versions de java (1.0 à 1.7) étaient centrées sur le paradigme orienté objet .

**A partir de la version 8 , le langage java s'est fortement inspiré du langage scala et est devenu un langage multi-paradigme "objet" et "fonctionnel" .**

Sans renier ses fondements orientés objets, le langage java a introduit à partir de la version 8 un **nouveau style complémentaire de programmation** appelée "**programmation fonctionnelle**" .

### 1.2. Programmation fonctionnelle

Au sein d'un contexte de programmation purement orienté objet , les objets comportent des données dont les valeurs (à l'instant t) représentent un certain état .

En appelant une méthode sur un objet , il y a souvent un changement de valeur(s) effectué et donc un changement d'état .

Exemple :

```
p1 = new Personne("jean", "Bon", 40) ;// où 40 signifie ici 40 ans
p1.incrementerAge() ;//premier changement d'état (40 ans → 41 ans)
p1.incrementerAge() ;//second changement d'état (41 ans → 42 ans)
System.out.println("nouvel age de p1=" + p1.getAge()) ; //affiche 42
```

A l'inverse en appliquant le style de programmation fonctionnelle , on manipule avant tout des fonctions pures (sans contexte , sans état contextuel) qui permettent d'appliquer des transformations (ou des calculs) .

Exemple :

```
Personne p1 = new Personne("jean","Bon",40);
System.out.println("p1="+p1);
Function<Personne,Personne> avecUnAnDePlus ;
avecUnAnDePlus = (Personne p) -> new Personne(p.getPrenom(),
                                p.getNom(),
                                p.getAge() + 1);
Personne p1Bis = avecUnAnDePlus.apply(avecUnAnDePlus.apply(p1));
//NB: l'objet (de données) p1 n'est pas modifié.
//la double application de la fonction de transformation avecUnAnDePlus
//a permis de créer un nouvel objet p1Bis ayant 2 ans de plus que p1.
System.out.println("p1Bis="+p1Bis);//42 ans
```



NB : La syntaxe java suivante

```
avecUnAnDePlus = (Personne p) -> new Personne(p.getPrenom(),  
                                              p.getNom(),  
                                              p.getAge() + 1);
```

est nouvelle depuis java 8 et correspond à une *lambda expression* .

Ça correspond ici à une fonction admettant en entrée un objet de type *Personne* et retournant en retour un autre objet *Personne* .

### 1.3. Programmation fonctionnelle accrochée à un squelette objet

Etant donné que java est un langage avant tout orienté objet , les nouveaux éléments ayant un style "fonctionnel" doivent s' accrocher à un squelette de code un peu orienté objet pour pouvoir exister , être compilé et être exécuté par une machine virtuelle java.

## 2. Lambda expressions et prog. fonctionnelle

### 2.1. Interfaces fonctionnelles , "lambda expression" et références

Une **interface fonctionnelle** (que l'on peut facultativement explicitement marquer 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);
}
...
}

```

## 2.2. Lambda expressions

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 .

Exemples (variations syntaxiques) :

```

Collections.sort(pList,
    (Person p1, Person p2) -> { return Integer.compare(p1.getAge() , p2.getAge()) ; })

```

ou bien avec return implicite (si une seule instruction dans { } ) :

```

Collections.sort(pList, (Person p1, Person p2) -> Integer.compare(p1.getAge() , p2.getAge()) )

```

ou bien (avec déduction/inférence des types de p1 et p2 selon le contexte) :

```
Collections.sort(pList, (p1, p2) -> Integer.compare(p1.getAge(), p2.getAge()))
```

Exemple dans son contexte :

```
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) */
        (Person p1, Person p2) -> Integer.compare(p1.getAge(), p2.getAge())
        /* (p1,p2) -> p1.getLastname().compareTo(p2.getLastname()) */
    );
    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);
}
```

## 2.3. Référence de fonctions

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);
```

## 2.4. 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(this.message); */
        Runnable r1 = () -> System.out.println(message);
        Thread t1 = new Thread(r1); t1.start();
    }
}
```

NB : En langage **java** , une lambda expression est formulée par  $(p) \rightarrow p.getXy() \geq \text{this.seuil}$   
en langage **javascript** , une fonction fléchée est formulée par  $(p) \Rightarrow p.xy \geq \text{this.seuil}$   
Dans les 2 cas le mot clef **this** (utilisé dans une lambda) fait référence au contexte objet englobant.

## 2.5. Function, Predicate , Supplier, Consumer, ...

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

<i>Interface Fonctionnelle</i>	<i>signature de la fonction</i>	<i>exemple</i>
<b>UnaryOperator</b> <T>	T apply(T t)	String::toLowerCase
<b>BinaryOperator</b> <T>	T apply(T t1, T, t2)	BigInteger::add
<b>Predicate</b> <T>	boolean test(T t)	Collection::isEmpty
<b>Function</b> <T, R>	R apply(T t)	Arrays::asList
<b>BiFunction</b> <T, U, R>	R apply(T t, U u)	(x, y) -> x * y
<b>Supplier</b> <T>	T get()	Instant::now
<b>Consumer</b> <T>	void accept(T t)	System.out::println
autres (variantes , ...)	...	....

Exemple d'utilisation (explicite et directe) :

```

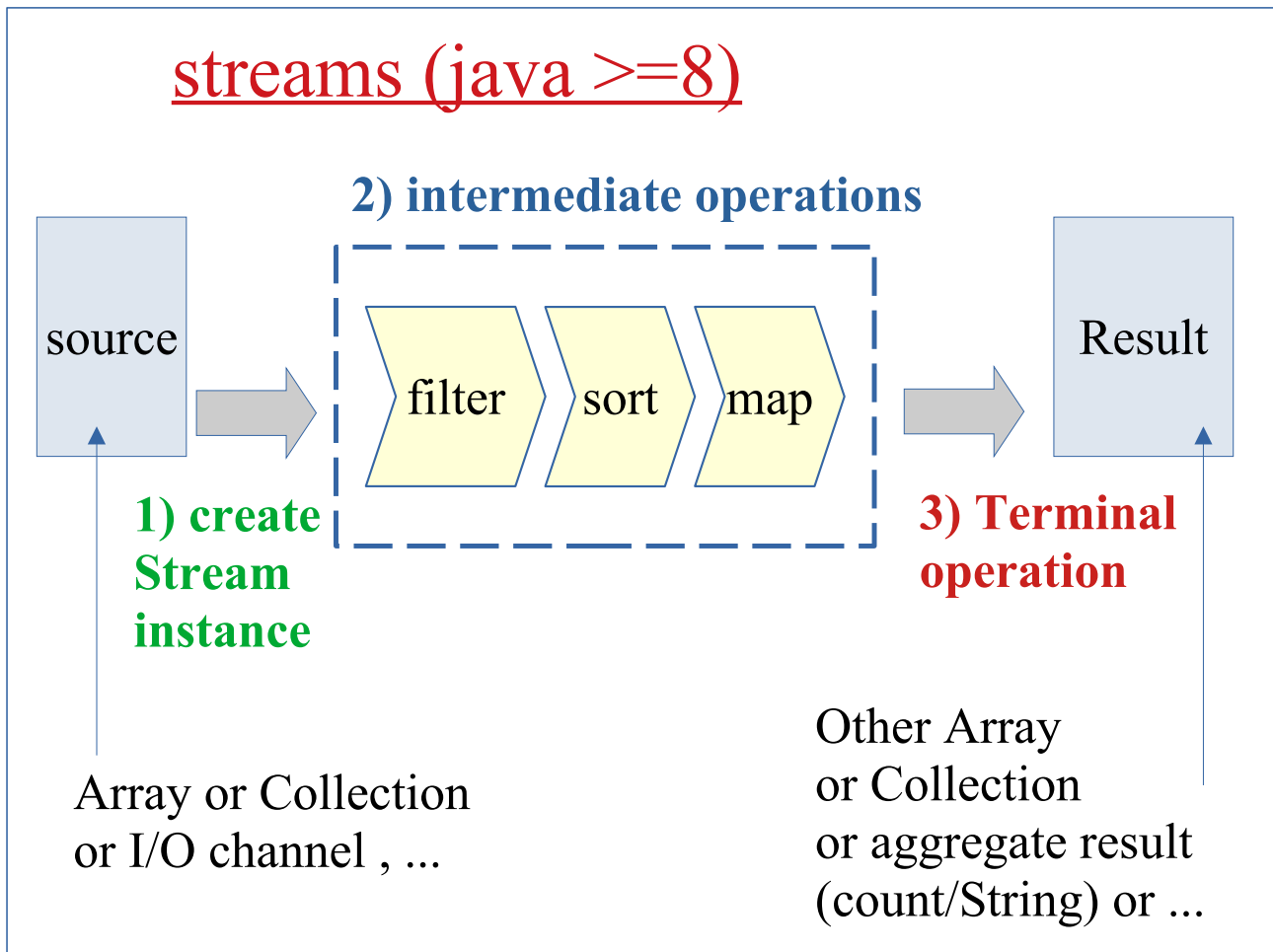
...
List<String> liste = initListOfString();
Function<String, String> atrFct = (name) -> {return "@" + name;};
Function<String, Integer> lengthFct = (name) -> name.length() ;
//ou bien Function<String, Integer> lengthFct = String::length ;
for (String s : liste) {
    System.out.println(atrFct.apply(s)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



### 3. Streams (java 8 et plus)



### 3.1. 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);
}
```

```
List<Person> listePersonnesFiltreesTrieesEtTransformees =
    persons.stream()
    .filter( (p)->p.getAge()>=18 )
    .sorted( (p1,p2)->Integer.compare(p1.getAge(), p2.getAge()))
    .map( (p) -> { p.setLastname(p.getLastname().toUpperCase()); return p; } )
    .collect(Collectors.toList());
```

```
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 exemples :

```
String s = IntStream.iterate(1, i -> i * 2) //generation d'une sequence a priori infinie 1, 2, 4, ...
    .limit(10) //limitation a 10 iterations
    .mapToObj(String :: valueOf) //transformation de chaque élément en String
    .collect(Collectors.joining(" ")); //collector rassemblant tout en une seule grande chaine
    //via des concatenations (en boucle) avec le separateur/jointeur spécifié
System.out.println(s);           //affiche 1 ; 2 ; 4 ; 8 ; 16 ; 32 ; 64 ; 128 ; 256 ; 512
```

```
List<Product> listProd = ProductUtil.initSampleProductList();
Optional<Product> ofp=listProd.stream().filter(p -> p.getPrice() >= 200).findFirst();
System.out.println("premier produit trouve avec prix >= 200 : " + ofp.orElse(null));

boolean auMoinsUnProduitExistantQuiCommenceParPrinter =
    listProd.stream().anyMatch(p -> p.getLabel().startsWith("printer"));
if(auMoinsUnProduitExistantQuiCommenceParPrinter)
    System.out.println("au moins un des produits existants commence par printer");
```

```
double prixMaxi =listProd.stream()
    .map(p -> p.getPrice())
    .reduce(Double::max).orElse(0.0);
System.out.println("prixMaxi="+prixMaxi);
```

```
List<String> liste1 = Arrays.asList("rouge", "vert", "bleu");
List<String> liste2 = Arrays.asList("jaune", "orange", "bleu");
List<String> listeCouleurs = Stream.concat(liste1.stream(), liste2.stream())
    .distinct()
    .collect(Collectors.toList());
System.out.println("liste des couleurs (sans doublon)="+listeCouleurs);
```

→ liste des couleurs (sans doublon)=[rouge, vert, bleu, jaune, orange]

```
List<Point> points = Arrays.asList(new Point(1, 2), new Point(2, 4));
System.out.println("liste de points =" + points);
points.stream().forEach(p -> p.translate(10, 5)); //forEach() = operation terminale (void)
    // p.translate() : void/ne retourne rien et modifie p
System.out.println("Apres translation(10,5), liste de points =" + points);
```

liste de points =[java.awt.Point[x=1, y=2], java.awt.Point[x=2, y=4]]  
 Apres translation(10,5), liste de points =[java.awt.Point[x=11,y=7], java.awt.Point[x=12,y=9]]

## VI - Classes utilitaires , aspects divers

### 1. Éléments 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 ici 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. Quelques classes utilitaires

### 2.1. 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
...
```

### 2.2. 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**).

## 2.3. 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. <b>setTime</b> (date /* de type <i>Date</i> 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.**HOURL\_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. <b>getTime</b> ();
---

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)



### 3. 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 française:*

```
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 **MyResources.properties** (ou **MyResources\_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
```

## 4. 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 , ....) .

### 4.1. DateFormat et SimpleDateFormat

```
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);

SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd");
String sdate = simpleDateFormat.format(new Date());
```

### 4.2. 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);
```

double simple quote  
(deux fois ')

### 4.3. 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));
```

## 5. Optional , DateTime , ...

### 5.1. 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 entre autres utilisé comme type de retour de certaines versions de `stream.reduce(...)` .

Depuis java 8 , le **type de retour `Optional<T>`** plutôt que T éventuellement null ne fait qu'**expliquer clairement** (*via le type , sans autre commentaire ou documentation supplémentaire*) **l'aspect optionnel d'une référence** vers un autre objet

Cela correspond à la **multiplicité 0..1** en UML

*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);
}

```

NB :

**optional.get();** retourne la valeur interne (non nulle) ou bien retourne une **exception** si vide/empty.  
**optional.orElse(null);** retourne la valeur interne (non nulle) ou bien retourne **null** si vide/empty.

```

public static void testDiversAvecNouvellesMethodesDeOptionalDepuisJava9() {
    Optional<String> opS1 = Optional.of("s1");
    Optional<String> opS2 = /*Optional.ofNullable(null); */ Optional.empty();

    //op.or(...) permet de construire et fournir un autre optional (valeur par défaut
    // ou plan B) si l'optionnel original est null/empty :
    System.out.println(opS1.or(()->Optional.of("default_string"))); //affiche Optional[s1]
    System.out.println(opS2.or(()->Optional.of("default_string")));
    //affiche Optional[default_string]

    //op.ifPresentOrElse(nomEmptyConsumer_as_lambda , emptyActionLambda)
    //permet de déclencher alternativement une lambda ou une autre en fonction d'un
    // contenu vide ou pas , pas d'enchaînement après (opération terminale en "void")
    opS1.ifPresentOrElse( (value) -> System.out.println("opS1="+value),
                        () -> System.out.println("value of opS1 is empty"));
    //affiche opS1=s1
    opS2.ifPresentOrElse( (value) -> System.out.println("opS2="+value),
                        () -> System.out.println("value of opS2 is empty"));
    //affiche value of opS2 is empty
}

```

## 5.2. 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);
}

```

```

LocalDateTime now = LocalDateTime.now();
System.out.println("basic/default display of LocalDateTime.now() :"+ now);
//ex :2021-11-23T09:28:12.289915600

Instant instantT = now.atZone(ZoneId.systemDefault()).toInstant();
long nbMsSinceFirstJanuary1970GMT = instantT.toEpochMilli();
System.out.println("instantT.toEpochMilli() , timestamp , nb ms since 1970-01-01 00:00:00
GMT="+ nbMsSinceFirstJanuary1970GMT); //ex : 1637656092289

```

```

LocalDate nowDate = LocalDate.now();
System.out.println("basic/default display : today (nowDate local) is " + nowDate); //ex :2021-11-23
String sdate_fr_2=nowDate.format(DateTimeFormatter.ofPattern("EEEE, dd MMMM
yyyy",Locale.FRENCH)); // EEEE means fullname of weekday
System.out.println("sdate_fr_2="+sdate_fr_2); //exemple: dimanche, 13 septembre 2020

```

### 5.3. 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();
}
}
```

## 5.4. 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) , ...

## VII - Entrées/sorties (io,nio,nio2) – fichiers

Pour manipuler , les entrées/sorties fichiers et le FileSystem , le langage java a apporté progressivement des librairies de classes au fur et à mesure des évolutions de versions.

**java.io** date de java 1 et fonctionne en mode synchrone bloquant.

**java.nio** de java 4 a apporté de nouvelles classes et abstractions.

**java.nio2** de java 7 a encore un peu amélioré les choses avec plus d'asynchronismes.

### 1. Essentiel de java.io

**NB:** Bien que le package **java.io** soit très ancien , il a le mérite de fonctionner avec toutes les versions de java (même sur android) et ses fonctionnalités (bien que réduites) peuvent suffirent pour des besoins simples.

#### 1.1. 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** / **FileOutputStream** 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:

<b>DataOutputStream</b>	Pour écrire <b>en binaire</b> des entiers, réels, ...
<b>PrintStream</b>	Pour écrire sous forme de <b>chaînes ASCII</b> des String,int,...
<b>BufferedReader &amp; InputStreamReader</b>	pour lire des chaînes de caractères que l'on récupérera sous forme de String,int,...

Les entrées/sorties standards (descripteurs linux 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** .

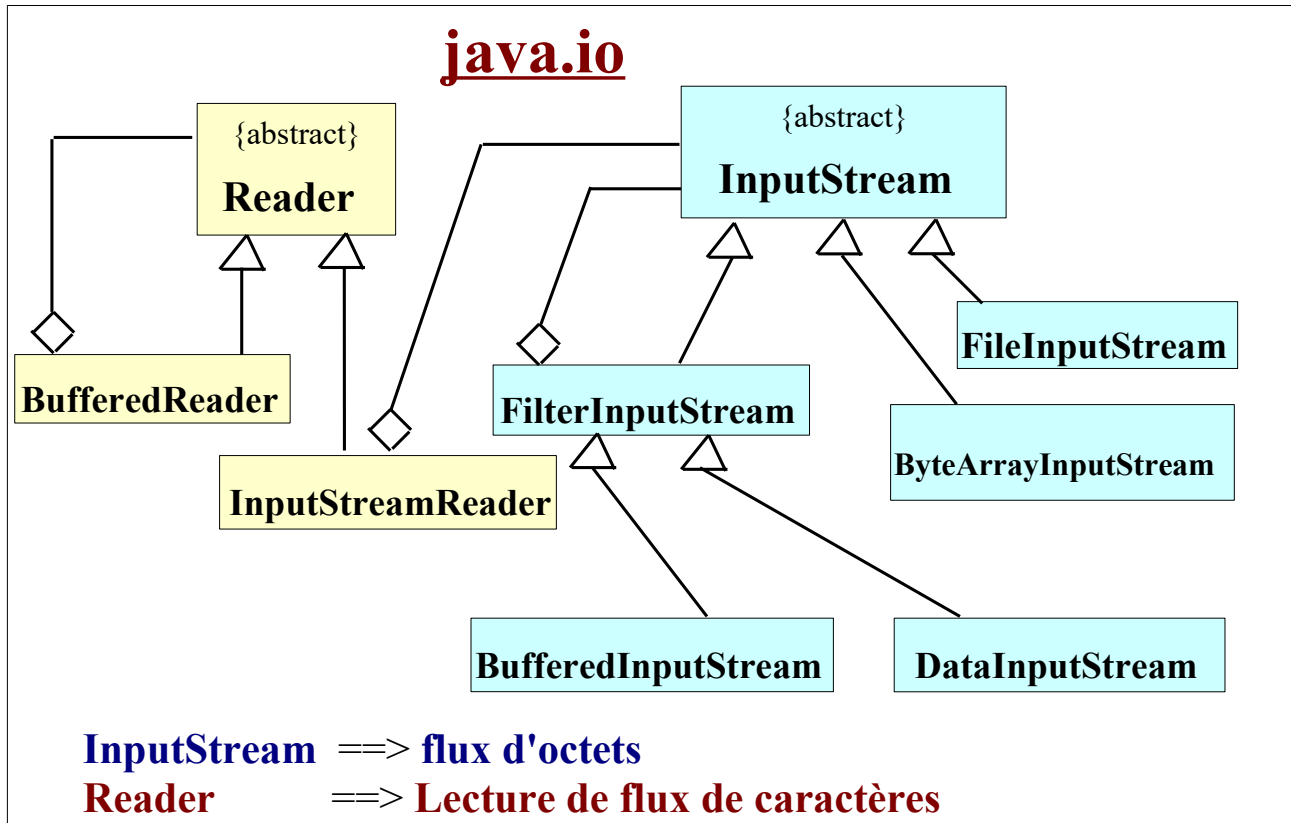
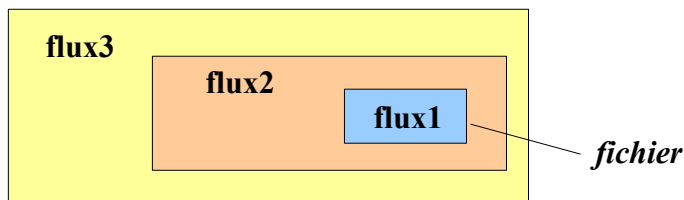
#### 1.2. Principe fondamental d'imbrication des flux

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");
```

Ce 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);
```

### 1.3. Viel exemple de code (possible depuis java 1)

Cet exemple permet de saisir une phrase à partir de l'entrée standard. Cette ligne est alors introduite dans un fichier fl.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 "fl.txt"

    FileOutputStream of = new FileOutputStream("fl.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("fl.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 */
}
```

## 1.4. Exemple 2 (lecture simple d'un petit fichier .csv)

*produits.csv*

```
numero;label;prix
1;cahier;5.5
2;stylo;2.1
3;classeur;6
4;gomme;3.3
```

```
...
public class Catalogue {

    private List<Produit> listeProduits = new ArrayList<>();

    public void lireFichier(String fileName) {
        try {
            FileInputStream fis = new FileInputStream(fileName);
            BufferedReader br = new BufferedReader(new InputStreamReader(fis));
            br.readLine();//lecture de la première ligne du fichier .csv avec titres colonnes

            String ligne=br.readLine();
            while(ligne != null) {
                String[] t = ligne.split(";");
                Produit p = new Produit(Integer.parseInt(t[0]),
                                         t[1],
                                         Double.parseDouble(t[2]));
                this.listeProduits.add(p);
                ligne=br.readLine();
            }
            br.close();fis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

//...
}
```

```
Catalogue c = new Catalogue();
c.lireFichier("produits.csv");
c.afficherProduits();
```

## 1.5. (java.io.File) répertoires et des attributs sur les fichiers

**Attention:** bien qu'encore utilisable, la classe **java.io.File** est beaucoup moins bien que **java.nio.Files** qui sera présentée ultérieurement en fin de chapitre.

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:

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

## 2. Spécificités à connaître et détails intéressants

### 2.1. java.util.zip

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();
```

### 2.2. Fichier de données accompagnant le code

Lorsqu'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).

## 3. Java.util.Scanner (depuis 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();
// Double d = reader.nextDouble();
// String ch = reader.next();
// String ligne = reader.nextLine();
```

`reader.close();` //à ne pas appeler trop tôt .

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

## 4. try with resources (AutoClosable)

Ce code opérationnel

```
public void lireFichierWithScanner(String fileName) {
    Scanner scanner = null;
    try {
        //scanner = new Scanner(new File(fileName));
        scanner = new Scanner(Paths.get(fileName));
        while(scanner.hasNext()) {
            String ligne = scanner.nextLine();
            System.out.println(ligne) ;
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    finally {
        if (scanner != null) {
            scanner.close();
        }
    }
}
```

peut se simplifier en

```
public void lireFichierWithScannerWithTryWithResources(String fileName) {
    //try (Scanner scanner = new Scanner(new File(fileName)) ){
    try (Scanner scanner = new Scanner(Path.of(fileName)) ){
        while(scanner.hasNext()) {
            String ligne = scanner.nextLine();
            System.out.println(ligne) ;
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    //Automatic finally with call
    //of .close() of AutoCloseable interface implementation in Scanner class
}
```

Variante possible depuis java9 :

```
final Scanner scanner = new Scanner(new File(fileName)) ;
try (scanner){ ...}
catch(...) { ... }
```

**try/catch avec plusieurs ressources :**

```
try (Scanner scanner = new Scanner(new File("testRead.txt")) ;  
    PrintWriter writer = new PrintWriter(new File("testWrite.txt"))) {  
    while (scanner.hasNext()) {  
        writer.print(scanner.nextLine());  
    }  
}
```

Avec ordre d'appels des .close() = inverse de l'ordre dans try(...)

**Ressource personnalisée compatible avec "try with resources" :**

```
public class MyResource implements AutoCloseable {  
    @Override  
    public void close() throws Exception {  
        System.out.println("Close MyResource");  
    }  
}
```

**Principales classes ou Interfaces Java ayant déjà implémenté l'interface AutoCloseable :**

- Scanner
- PrintWriter et beaucoup d'autres Writer
- Reader , InputStreamReader , BufferedReader, ....
- InputStream , FileInputStream , ObjectInputStream, ...
- OutputStream , FileOutputStream , DataOutputStream , ObjectOutputStream, ....
- Pipe, Socket , ...
- Connection, Statement , ResultSet , RowSet de jdbc

## 5. NIO2 (new IO)

### 5.1. Introduction à nio , nio2

Dès l'origine du langage java , la classe File du package java.io était disponible pour accéder aux éléments du "file system" (fichier , répertoire, ...) et pour effectuer des opérations dessus (tester l'existence d'un fichier , supprimer des fichiers , ...).

Bien qu'opérationnel , le package java.io (et la classe File) ont quelques limitations importantes :

- La classe File manque de fonctionnalité importante telle qu'une méthode "copy".
- Beaucoup de méthode retourne des booléens plutôt que des exceptions . Ce qui rend assez délicate l'analyse des problèmes (raison exacte de l'erreur?).
- Pas de bonne gestion des liens symboliques .
- Un ensemble très limité d'attributs sur les fichiers sont disponibles sur java.io.File.

Pour surmonter certaines lacunes , un nouveau package "java.nio" a été introduit dès java4 de façon à apporter quelques améliorations très techniques telles que les suivantes :

- *Channels et Selectors*: un "channel" est une abstraction d'une caractéristiques de bas niveau telle qu'un fichier mappé en mémoire .
- *Buffers*: Buffering pour toutes les classes wrapper/primitives (sauf pour Boolean).
- *Jeu de caractères*: **Charset** (*java.nio.charset*), avec encodeurs et décodeurs entre bytes[] et symboles "Unicode".

La version 7 de java a introduit un nouveau (gros) package "**java.nio.file**" (alias nio2) .

Ce package apporte à la fois de nouvelles fonctionnalités techniques telles que :

- gestion des liens symboliques
- bonne gestion des "Path" (aspect bien séparé).
- meilleur gestion des "File attributes"

et de nouvelles syntaxes (plus concises , plus "orienté objet") telles que les remontées d'exceptions.

Attention : Entre java1, java4 , java7 , des ajouts , mais pas de suppression (pour garder une compatibilité avec les anciennes versions).

Bien que la classe "java.io.File" n'est pas officiellement considérée comme obsolète / "deprecated" , il est très conseillé d'utiliser les nouvelles classes de nio2 (sachant qu'il existe des passerelles entre "java.io" et "java.nio.file") .

## 6. Principales classes et interfaces de nio2

**NIO 2** (associé aux packages "**java.nio.file**" et "**java.nio.file.attribute**") repose sur plusieurs classes et interfaces dont les principales sont :

- **Path** : encapsule un chemin dans le système de fichiers
- **Files** : avec méthodes statiques pour manipuler les éléments du système de fichiers
- *FileSystemProvider* : fournisseur technique (implémentation) de FS



- **FileSystem** : encapsule un système de fichiers
- **FileSystems** : fabrique permettant entre autres de créer une instance de FileSystem
- **FileStorage** : système de stockage (ex : partition / volume logique , ....)

**Quelques équivalences/transpositions de fonctionnalités basiques entre "java.io" et "nio2" :**

Fonctionnalité	java.io	NIO 2
Encapsuler un chemin	java.io.File	java.nio.file.Path
Vérifier les permissions	File.canRead(), File.canCrite() et File.canExecute()	Files.isReadable(), Files.isWritable() et Files.isExecutable().
Vérifier le type d'élément	File.isDirectory(), File.isFile()	Files.isDirectory(Path, LinkOption...), Files.isRegularFile(Path, LinkOption...),
Taille d'un fichier	File.length()	Files.size(Path)
Obtenir ou modifier la date de dernière mise à jour	File.lastModified() , File.setLastModified(long)	Files.getLastModifiedTime(Path, LinkOption...), Files.setLastModifiedTime(Path, FileTime)
Modifier les attributs	File.setExecutable(), File.setReadable(), File.setReadOnly(), File.setWritable()	Files.setAttribute(Path, String, Object, LinkOption...)
Déplacer un fichier	File.renameTo()	Files.move()
Supprimer un fichier	File.delete()	Files.delete()
Créer un fichier	File.createNewFile()	Files.createFile()
Créer un fichier temporaire	File.createTempFile()	Files.createTempFile(Path, String, FileAttributes<?>), Files.createTempFile(Path, String, String, FileAttributes<?>)
Tester l'existence d'un fichier	File.exists	Files.exists() ou Files.notExists()
Obtenir le chemin absolu	File.getAbsolutePath() ou File.getAbsolutePath()	Path.toAbsolutePath()
Chemin canonique (sans ".", ".." )	File.getCanonicalPath() ou File.getCanonicalFile()	Path.toRealPath() ou Path.normalize()
Convertir en URI	File.toURI()	Path.toURI()
L'élément est-il caché	File.isHidden()	Files.isHidden()
Obtenir le contenu d'un répertoire	File.list() ou File.listFiles()	Path.newDirectoryStream()
Créer un répertoire	File.mkdir() ou File.mkdirs()	Path.createDirectory()
Obtenir le contenu du répertoire racine	File.listRoots()	FileSystem.getRootDirectories()
Place totale , libre , ... sur FS	File.getTotalSpace() File.getFreeSpace()	FileStore.getTotalSpace() FileStore.getUnallocatedSpace()

## 7. Gestion des chemins (Path)

L'interface "java.nio.file.**Path**" est une représentation abstraite d'un chemin (relatif ou absolu) au sein d'un système de fichiers.

Un chemin peut référencer un fichier , un répertoire , un lien symbolique , un sous-chemin , ...

Les instances de "Path" sont "immuables" et peuvent être utilisées dans un contexte multi-threads.

### 7.1. Obtention d'une instance de "Path"

Récupération d'un chemin en appelant explicitement **getPath()** sur une instance de "**FileSystem**" :

```
Path chemin = FileSystems.getDefault().getPath(
    "C:/Users/powerUser/Temp/monfichier.txt");
```

équivalent indirect via la méthode (utilitaire / "helper") statique **Paths.get()** :

```
Path chemin = Paths.get("C:/Users/powerUser/Temp/monfichier.txt");
Path chemin2 = Paths.get(URI.create("file:///C:/Users/powerUser/Temp/monfichier.txt"));
Path chemin3 = Paths.get(System.getProperty("java.io.tmpdir"), "monfichier.txt");
```

NB : **Path.of()** existe depuis java 11 et se comporte comme **Paths.get()**

NB : bien que (sous windows) on puisse utiliser le séparateur "\" (ex : [C:\\RepXy\\monfichier.txt](#)) , il vaut mieux utiliser le séparateur "/" (plus portable et plus simple) .

*Passerelle "io / nio2" (depuis java 7) :*

En partant d'une instance de l'ancienne classe **java.io.File** , on peut appeler la nouvelle méthode **".toPath()"** pour récupérer une instance de "java.nio.file.Path".

### 7.2. Obtention des éléments d'un chemin

Méthode	Rôle
String <b>getFileName()</b>	Retourne le nom du dernier élément du chemin. Si le chemin concerne un fichier alors c'est le nom du fichier qui est retourné
Path <b>getName(int index)</b>	Retourne l'élément du chemin dont l'index est fourni en paramètre. Le premier élément (hors racine) possède l'index 0 et correspond à la première partie sous la racine (ex : "windows" sous racine " <a href="#">c:/</a> ").
int <b>getNameCount()</b>	Retourne le nombre d'éléments du chemin (hors racine)
Path <b>getParent()</b>	Retourne le chemin parent ou null s'il n'existe pas
Path <b>getRoot()</b>	Retourne la racine d'un chemin absolu (par exemple C:\ sous windows ou / sous Unix) ou null pour un chemin relatif
String <b>toString()</b>	Retourne le chemin sous la forme d'une chaîne de caractères
Path <b>subPath(int beginInclusiveIndex, int endExclusiveIndex)</b>	Retourne un sous-chemin (hors racine) [beginIndex,endIndex[

Exemple avec un <b>chemin absolu</b> :	Exemple avec un <b>chemin relatif</b> :
<b>path=C:\Windows\Fonts\arial.ttf</b>  toString() = C:\Windows\Fonts\arial.ttf getFileName() = arial.ttf getRoot() = C:\ getName(0) = Windows getNameCount() = 3 getParent() = C:\Windows\Fonts subpath(0,2) = Windows\Fonts	<b>path=Fonts\arial.ttf</b>  toString() = Fonts\arial.ttf getFileName() = arial.ttf getRoot() = null getName(0) = Fonts getNameCount() = 2 getParent() = Fonts subpath(0,2) = Fonts\arial.ttf

### 7.3. Manipulation , combinaison et conversions de chemins

Méthode	Rôle
Path <b>normalize()</b>	Normaliser (ou rendre "canonique") un chemin en supprimant les éléments non indispensables « . » et « .. » qu'il contient.
Path <b>relativize</b> (Path other)	Retourner le chemin relatif permettant d'aller du "path courant" vers celui fourni en paramètres .
Path <b>resolve</b> (Path relative)	Combiner deux chemins (courant + relatif_en_arg) pour former global
Path <b>toAbsolutePath()</b>	Retourne un <b>chemin absolu</b> (en <i>tenant compte du répertoire courant du contexte d'exécution</i> (idem à "pwd")).
Path <b>toRealPath</b> (LinkOption...)	Retourner le <b>chemin physique</b> du chemin notamment en <b>résolvant les liens symboliques selon les options fournies</b> . Peut lever une exception si le fichier au bout du chemin courant (absolu ou relatif) n'existe pas .
URI <b>toUri()</b>	Retourner le chemin sous la forme d'une URI ( <a href="#">file:///</a> )

#### Exemple1 :

```
path.toString() = C:\Utilisateurs\...\Windows\...\Fonts\arial.ttf
path.normalize() = C:\Windows\Fonts\arial.ttf
```

```
path.toString() = ..\..\Fonts\..\arial.ttf
path.normalize() = ..\..\Fonts\arial.ttf (seul le ../ inutile supprimé)
```

#### Exemple2 :

```
Path windowsFontPath = Paths.get("c:/Windows/Fonts");
Path usersPath = Paths.get("c:/Utilisateurs");
//relative path from current to arg
Path relativePathFromUsersToWindowsFonts=
usersPath.relativize(windowsFontPath);
System.out.println("relative path=" + relativePathFromUsersToWindowsFonts) ;

relative path=..\Windows\Fonts
```

#### Exemple3 :

```
Path windowsPath = Paths.get("c:/Windows");
Path arialRelativePath = Paths.get("Fonts/arial.ttf");
Path globalPath = windowsPath.resolve(arialRelativePath);
System.out.println("globalPath="+globalPath);

globalPath=c:\Windows\Fonts\arial.ttf
```

```
Path windowsFontsPath = Paths.get("C:/Windows/Fonts");
System.out.println("uri=" + windowsFontsPath.toUri()); //java.net.URI

uri=file:///C:/Windows/Fonts/
```

```
Path relativePath = Paths.get(".");
Path absolutePath = relativePath.toAbsolutePath(); //selon rep courant (pwd)
System.out.println("absolutePath=" + absolutePath);

absolutePath=C:\Users\didier\workspace\test_j7_j8\.
```

Rappel : sous linux , un lien symbolique se construit via la commande  
**\$ ln -s /nom\_du\_dossier\_source nom\_du\_lien**

**Attention, sous windows , un lien symbolique ne doit pas être confondu avec un raccourci .**  
 A l'époque de windows XP , un lien symbolique se construisait avec la commande "*junction.exe*" (à installer).

Depuis "Vista" , et encore aujourd'hui avec windows 7, 8 et 10, un lien symbolique se construit avec la commande **mklink** suivante :

#### **MKLINK [/D] | [/H] | [/J] Lien Cible**

**/D** : Crée un lien symbolique vers un répertoire. Par défaut, il s'agit d'un lien symbolique vers un fichier.

**/H** : Crée un lien réel à la place d'un lien symbolique.

**/J** : Crée une jonction de répertoires.

**Lien** : Spécifie le nom du nouveau lien symbolique.

**Cible** : Spécifie le chemin d'accès (relatif ou absolu) auquel le nouveau lien fait référence.

Exemple (à lancer avec des droits "administrateur"):

```
cd c:\tmp\bb
```

```
mklink /J lien_vers_aa c:\tmp\aa
```

jonction créée pour lien\_vers\_aa <==> c:\tmp\aa

```
Path path = Paths.get("C:/tmp/bb/raccourci_vers_aa/f1.txt");
→ java.nio.file.NoSuchFileException: C:\tmp\bb\raccourci_vers_aa\f1.txt
```

```
Path path = Paths.get("C:/tmp/bb/lien_vers_aa/f1.txt");
try {
    System.out.println("chemin indirect sans suivre résolution lien symbolique="
        + path.toRealPath(LinkOption.NOFOLLOW_LINKS));
    System.out.println("chemin direct suivant résolution lien symbolique="
        + path.toRealPath());
} catch (IOException e) {
    e.printStackTrace();
}

chemin indirect sans suivre résol lien symbolique=C:\tmp\bb\lien_vers_aa\f1.txt
chemin direct suivant résolution lien symbolique=C:\tmp\aa\f1.txt
```

## 7.4. Comparaisons de chemins

La méthode `.equals()` permet de tester si deux chemins ont des valeurs identiques.

L'interface **Path** hérite de **Comparable**. Les chemins peuvent donc être automatiquement **triés**.

L'interface **Path** comporte en outre les méthodes de comparaison spécifiques suivantes :

Méthode	Rôle
int <b>compareTo</b> (Path other) (Comparable<Path>)	Comparer le chemin avec celui fourni en paramètre et retourne 0 si identiques , <0 avant , >0 si après
boolean <b>endsWith</b> (Path other) boolean <b>endsWith</b> (String other)	Comparer la fin du chemin avec celui fourni en paramètre
boolean <b>startsWith</b> (Path other) boolean <b>startsWith</b> (String other)	Comparer le début du chemin avec celui fourni en paramètre

## 7.5. interface PathMatcher avec méthode matches(path) et "glob"

Un **"glob"** est une expression basée sur certains méta-caractères qui seront comparés à des parties de "path" .

```
Path path = Paths.get("C:/tmp/aa/fl.txt");
PathMatcher txtMatcher = FileSystems.getDefault().getPathMatcher("glob:*.txt");
if (txtMatcher.matches(path.getFileName())) {
    System.out.println(path + " reference un fichier texte");
}
```

Motif (dans glob)	Rôle associé / correspondance
*	Aucun ou plusieurs caractères
**	Aucun ou plusieurs sous-répertoires
?	Un caractère quelconque
{ }	Un ensemble de motifs exemple : {htm, html}
[ ]	Un ensemble de caractères. Exemple : [A-Z] : toutes les lettres majuscules [0-9] : tous les chiffres [a-z,A-Z] : toutes les lettres indépendamment de la casse Chaque élément de l'ensemble est séparé par un caractère virgule Le caractère - permet de définir une plage de caractères A l'intérieur des crochets, les caractères *, ? et / ne sont pas interprétés
\	Il permet d'échapper des caractères pour éviter qu'ils ne soient interprétés.

	Il sert notamment à échapper le caractère \ lui-même
Les autres caractères	Ils se représentent eux-mêmes sans être interprétés

Quelques exemples :

<i>Glob</i>	<i>Correspondances</i>
<b>*.html</b>	tous les fichiers ayant l'extension .html
<b>???</b>	trois caractères quelconques
<b>*[0-9]*</b>	tous les fichiers qui contiennent au moins un chiffre
<b>*.{htm, html}</b>	tous les fichiers dont l'extension est htm ou html
<b>Test*.java</b>	tous les fichiers dont le nom commence par un Test et possède une extension .java

**NB :** L'interface **PathMatcher** (existant depuis java 7) comporte une unique méthode "*Boolean matches(Path path)*". Elle peut donc être utilisée au sein de "lambda expression" depuis java 8.

Exemple (fonctionnant depuis java8) :

...  
....

## 8. Classe utilitaire Files ("*helper*" avec 50 méthodes statiques)

### 8.1. Vérifications sur fichiers ou répertoires

Méthode	Rôle
boolean <b>exists</b> (Path)	vérifier l'existence sur le système de fichiers de l'élément dont le chemin est encapsulé dans le paramètre de type Path fourni
boolean <b>notExists</b> (Path)	
boolean <b>isReadable</b> (Path path)	peut être lu (droits en lecture) ?
boolean <b>isWritable</b> (Path path)	peut être modifié (droits en écriture) ?
boolean <b>isHidden</b> (Path path)	est caché ?
boolean <b>isExecutable</b> (Path path)	est exécutable ?
boolean <b>isRegularFile</b> (Path path)	est un fichier ?
boolean <b>isDirectory</b> (Path path)	est un répertoire ?
boolean <b>isSymbolicLink</b> (Path path)	est un lien symbolique ?
String <b>probeContentType</b> (Path path)	retourne type MIME d'un fichier (ex: "text/plain") (ou null si indéterminé). S'appuie par défaut sur l'OS sous jacent.

Exemple:

```
Path path = Paths.get("C:/tmp/aa/fl.txt");
if(Files.isRegularFile(path)){
    System.out.println(path + " est un chemin vers un fichier");
}
```

## 8.2. Création d'un fichier ou d'un répertoire

Méthode	Rôle ou bien exemple
Path <b>createFile</b> (Path path, FileAttribute<?>... attrs)	Créer un fichier dont le chemin est encapsulé par l'instance de type Path fournie en paramètre
Path <b>createDirectory</b> (Path dir, FileAttribute<?>... attrs)	Path monRepertoire = Paths.get("C:/temp/mon_repertoire"); <b>Files.createDirectory</b> (monRepertoire);
Path <b>createDirectories</b> (Path dir, FileAttribute<?>... attrs)	Créer d'un seul coup plusieurs niveaux de sous répertoire selon le chemin exprimé "C:/temp/ <b>niveau1/niveau2/niv3</b> "
Path <b>createTempDirectory</b> (Path dir, String <b>prefix</b> , FileAttribute<?>... attrs) Path <b>createTempDirectory</b> (String <b>prefix</b> , FileAttribute<?>... attrs)	Créer un répertoire temporaire de nom " <b>prefixe indiqué</b> " + numéro_calculé_par_syst (ex : <b>rep_1245643</b> ) au sein du répertoire indiqué ou (à défaut) au sein du répertoire système (par défaut) prévu pour les temporaires.
Path <b>createTempFile</b> (Path dir, String <b>prefix</b> , String <b>suffix</b> , FileAttribute<?>... attrs) Path <b>createTempFile</b> (String <b>prefix</b> , String <b>suffix</b> , FileAttribute<?>... attrs)	Créer un fichier temporaire de nom " <b>prefixe indiqué</b> " + numéro_calculé_par_syst + " <b>suffixe indiqué</b> " (ex : <b>fic_1245643.txt</b> ) au sein du répertoire indiqué ou (à défaut) au sein du répertoire système (par défaut) prévu pour les temporaires. Le suffix peut éventuellement être à null .

Les **attributs** (de type *FileAttribute*) sont **facultatifs** (des valeurs par défaut existent).

Les attributs possibles seront étudiés ultérieurement .

**NB** : **createDirectory()** créer un seul niveau de sous répertoire à la fois (*contrairement à createDirectories*) .

Si le répertoire existe déjà , l'exception *FileAlreadyExists* est remontée.

Si le répertoire parent n'existe pas , *NoSuchFileException* est remontée.

## 8.3. Copie d'un fichier ou d'un répertoire

Méthode	Rôle / fonctionnalité
Path <b>copy</b> (Path <b>source</b> , Path <b>target</b> , CopyOption... options)	Copier un élément avec les options précisées ( <i>StandardCopyOption.REPLACE_EXISTING</i> , <i>StandardCopyOption.COPY_ATTRIBUTES</i> )
long <b>copy</b> ( <i>InputStream</i> in, Path <b>target</b> ,	Copier tous les octets d'un flux de type



CopyOption... options)	InputStream vers un fichier
long <b>copy</b> (Path source, <b>OutputStream</b> out)	Copier tous les octets d'un fichier dans un flux de type OutputStream

**NB :** L'option pointue LinkOption.NOFOLLOW\_LINKS permet de recopier si besoin un lien symbolique au bout du path indiqué (plutôt que de suivre le lien).

**NB2:** il est possible d'utiliser la méthode copy() sur un répertoire . Cependant, le répertoire sera créé sans que les fichiers contenus et .. ne soient eux aussi copiés .

Quoi que contienne le répertoire, la méthode copy ne crée qu'un répertoire vide. Pour copier le contenu du répertoire, il faut parcourir son contenu et copier chacun des éléments un par un.

## 8.4. Déplacement et suppression d'un fichier ou d'un répertoire

Méthode	Rôle
<b>move</b> (Path source, Path target, CopyOption... options)	Déplacer ou renommer un élément avec les options précisées (StandardCopyOption.REPLACE_EXISTING , StandardCopyOption.ATOMIC_MOVE )
void <b>delete</b> (Path path)	Supprimer un élément du système de fichiers (avec exception s'il n'existe pas ou si répertoire pas vide)
boolean <b>deleteIfExists</b> (Path path)	Supprimer un élément du système de fichiers s'il existe (sans exception s'il existe pas)

Si un déplacement efficace/performant demandé en mode "ATOMIC\_MOVE" est impossible (par exemple déplacement de "[c:/repXy](#)" vers "[d:/repXy](#)" ), une exception est alors remontée. On peut alors éventuellement ré-essayer sans l'option .

Des exceptions peuvent potentiellement remonter si un répertoire à déplacer n'est pas vide et que certains fichiers contenus sont en cours d'utilisation.

## 9. Parcours des éléments d'un répertoire

La solution de parcours proposée par NIO2 est plus performante que java.io.File.list(...) .

La méthode **newDirectoryStream()** de la classe utilitaire **Files** attend en paramètre un objet de type Path qui correspond au répertoire à parcourir et permet d'obtenir une instance de "**stream**" de type **DirectoryStream<Path>** (à parcourir avec un itérateur ou autre) .

Attention: il est très important d'invoquer la méthode **close()** de l'instance de type **DirectoryStream** pour libérer les ressources utilisées.

Exemple :

```
Path tmpAaPath = Paths.get("C:/tmp/aa");
DirectoryStream<Path> stream = null;
try {
    stream=Files.newDirectoryStream(tmpAaPath);
    Iterator<Path> iterator = stream.iterator();
    while(iterator.hasNext()) {
```



```

        Path p = iterator.next();
        System.out.println(p);
    }

    catch(IOException ex){    ex.printStackTrace();
    }
    finally {
        try {stream.close();
        } catch (IOException e) {e.printStackTrace();
        }
    }
}

```

Exemple amélioré et simplifié :

```

Path tmpAaPath = Paths.get("C:/tmp/aa");

//NB1 : le second paramètre facultatif de Files.newDirectoryStream()
// sert à filtrer les éléments à parcourir (sans besoin de préfixe glob:)

//NB2: L'interface DirectoryStream implémente hérite de Closable et
//le try(avec_ressource_implémentant_interface Closable)
//sera automatiquement associé à un finally implicite déclenchant .close()

try (DirectoryStream<Path> stream = Files.newDirectoryStream(tmpAaPath,"*.txt")){
    for(Path p : stream){
        System.out.println(p);
    }
}
catch(IOException ex){
    ex.printStackTrace();
}

```

On peut également paramétrer et utiliser un filtre spécifique lors du parcours :

```

Path tmpAaPath = Paths.get("C:/tmp/aa");
DirectoryStream.Filter<Path> littleSizeFilter = new DirectoryStream.Filter<Path>() {
    public static final long MEGABYTE = 1024*1024;
    @Override
    public boolean accept(Path element) throws IOException {
        return (Files.size(element) <= MEGABYTE );
    }
}; //fin de classe anonyme imbriquée implémentant DirectoryStream.Filter<Path>

try (DirectoryStream<Path> stream = Files.newDirectoryStream(tmpAaPath,littleSizeFilter)){
    for(Path p : stream){
        System.out.println(p);
    }
}

```

NB : ce code (java7) pourra être amélioré/simplifié via une lambda expression de java8 .

## 10. Parcours d'une hiérarchie de répertoires (visiteur)

La méthode **Files.walkFileTree()** permet de parcourir une (sous-)arborescence de répertoires en utilisant le **design pattern "visiteur"**. Ce type de parcours peut être utilisé pour rechercher, copier, déplacer, supprimer, ... des éléments de la hiérarchie parcourue.

Il faut écrire une classe qui implémente l'interface **java.nio.file.FileVisitor<T>**. Cette interface définit des méthodes qui seront des callbacks appelées lors du parcours de la hiérarchie.

Méthode	Rôle / fonctionnalité
<b>FileVisitResult postVisitDirectory</b> (T dir, IOException exc)	Le parcours sort d'un répertoire qui vient d'être parcouru ou une exception est survenue durant le parcours
<b>FileVisitResult preVisitDirectory</b> (T dir, BasicFileAttributes attrs)	Le parcours rencontre un répertoire, cette méthode est invoquée avant de parcourir son contenu
<b>FileVisitResult visitFile</b> (T file, BasicFileAttributes attrs)	Le parcours rencontre un fichier
<b>FileVisitResult visitFileFailed</b> (T file, IOException exc)	La visite d'un des fichiers durant le parcours n'est pas possible et une exception a été levée

Les méthodes de l'interface **FileVisitor** renvoient toutes une valeur qui appartient à l'énumération **FileVisitResult**. Cette valeur permet de contrôler le processus de parcours de l'arborescence :

- **CONTINUE** : poursuite du parcours
- **TERMINATE** : arrêt immédiat du parcours
- **SKIP\_SUBTREE** : inhibe le parcours de la sous-arborescence.
- **SKIP\_SIBLING** : inhibe le parcours des répertoires frères.

*Exemple(s) à ajouter ici plus tard .*

## 11. FileSystem (par défaut et "personnalisé")

### 11.1. Fabrique FileSystems

**FileSystems** est une **fabrique** (avec **méthodes statiques**) pour obtenir des objets **FileSystem**.

**.getDefault()** renvoie l'instance de type **FileSystem** qui encapsule le F.S. de la JVM.

**.getFileSystem(fsUri)** renvoie un **FileSystem** selon l'URI est fourni en paramètre.

**.newFileSystem()** surchargée permet de créer une instance spécifique de type **FileSystem** (cas pointu)

### 11.2. FileSystem

String separator = FileSystems.getDefault().**getSeparator()**; // / sous linux ou \ sous windows

```

Iterable<Path> dirs = FileSystems.getDefault().getRootDirectories();
for (Path name: dirs) {
    System.err.println(name); // C:\ , D:\ , ...
}

```

### 11.3. FileSystem spécifique

...

## 12. Lecture et écriture dans un fichier

### 12.1. Vue d'ensemble

Principaux apports de NIO et NIO2 :

IO	NIO	NIO2
Depuis Java 1.0 et 1.1	Depuis <b>Java 1.4</b> (JSR 151)	Depuis <b>Java 7</b> (JSR 203)
Synchrone bloquant	Synchrone non bloquant	Asynchrone non bloquant
File <b>InputStream</b> <b>OutputStream</b> Reader (Java 1.1) Writer (Java 1.1) Socket RandomAccessFile	<b>FileChannel</b> SocketChannel ServerSocketChannel (Charset, Selector, ByteBuffer)	Path <b>AsynchronousFileChannel</b> <b>AsynchronousByteChannel</b> AsynchronousSocketChannel AsynchronousServerSocketChannel SeekableByteChannel

### 12.2. Options sur l'ouverture d'un fichier

L'énumération **StandardOpenOption** implémente l'interface *OpenOption* et définit les options d'ouverture standard d'un fichier :

Valeur	Signification
APPEND	Si le fichier est ouvert en écriture alors les données sont ajoutées au fichier. Cette option doit être utilisée avec les options CREATE ou WRITE
CREATE	Créer un nouveau fichier s'il n'existe pas sinon le fichier est ouvert
CREATE_NEW	Créer un nouveau fichier : si le fichier existe déjà alors une exception est levée

DELETE_ON_CLOSE	Supprimer le fichier lorsque son flux associé est fermé : cette option est utile pour des fichiers temporaires
DSYNC	Demander l'écriture synchronisée des données dans le système de stockage sous-jacent (pas d'utilisation des tampons du système) ???
READ	Ouvrir le fichier en lecture
SPARSE	Indiquer au système que le fichier est clairsemé ce qui peut lui permettre de réaliser certaines optimisations si l'option est supportée par le système de fichiers (c'est notamment le cas avec NTFS)
SYNC	Demander l'écriture synchronisée des données et des métadonnées dans le système de stockage sous-jacent
TRUNCATE_EXISTING	Si le fichier existe et qu'il est ouvert en écriture alors il est vidé. Cette option doit être utilisée avec l'option WRITE
WRITE	Ouvrir le fichier en écriture

### 12.3. Lecture de l'intégralité d'un fichier / Files.readAllLines()

***Lecture (en boucle) de toutes les lignes d'un fichier texte :***

```
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
...
List<String> lignes = Files.readAllLines(
    FileSystems.getDefault().getPath("c:/tmp/aa/fl.txt"), StandardCharsets.UTF_8);
for (String ligne : lignes)
    System.out.println(ligne);
```

***Lecture d'un bloc de tout un (petit) fichier binaire :***

```
byte[] binaryContent = Files.readAllBytes(binaryFilePath);
```

# VIII - Introspection et Sérialisation

## 1. Introspection (java.lang.reflect)

### 1.1. Potentiel de l'intropection

#### 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++) ..."
```

Deux grandes variantes du point de départ :

```
Class<?> c = Class.forName("nomPackage.NomClasseJava") ;
```

*ou bien*

```
Class<?> c = objectInstance.getClass() ;
```

java.lang.**Class**<T> est une **méta-classe** : une classe très spéciale dont une instance décrit la structure d'une classe ordinaire.

## 1.2. Exemple d'introspection

Cet exemple montre comment générer automatiquement des lignes de fichier .csv depuis un objet java quelconque (Devise, Produit, Strat, Personne, ...) :

```
package util;
import java.io.FileOutputStream; import java.io.IOException;
import java.io.PrintStream; import java.lang.reflect.Field;
import java.util.List;

public class MyCsvUtil {

    public static String writeValuesAsCsvString(Object obj){
        String csvString=null;
        Class<?> c = obj.getClass(); //meta description de la classe de l'objet obj
        try {
            Field[] tabChamps = c.getDeclaredFields();
            for(Field f : tabChamps) {
                f.setAccessible(true); //pour accéder aux parties privées
                String fieldValue= f.get(obj)==null?null:f.get(obj).toString();
                csvString=(csvString==null)?fieldValue : csvString + ";" +fieldValue;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return csvString;
    }

    public static String writeFieldNamesAsCsvString(Object obj){
        String csvString=null;
        Class<?> c = obj.getClass(); //meta description de la classe de l'objet obj
        try {
            Field[] tabChamps = c.getDeclaredFields();
            for(Field f : tabChamps) {
                String fieldName= f.getName();
                csvString=(csvString==null)?fieldName:csvString + ";" +fieldName;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return csvString;
    }

    public static void writeValuesAsCsvFile(List<?> col,String fileName){
        if(col.isEmpty()) return;//fin fonction sans rien faire
        try (PrintStream ps = new PrintStream(new FileOutputStream(fileName))){
            Object firstObj = col.get(0);
            String ligneEnteteAuFormatCsv= writeFieldNamesAsCsvString(firstObj);
            ps.println(ligneEnteteAuFormatCsv);
            for(Object obj:col) {
                String ligneValeursAuFormatCsv= writeValuesAsCsvString(obj);
                ps.println(ligneValeursAuFormatCsv);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        //try_with_autoCloseable resource
    }
}
```

Le code ci dessus analyse automatiquement les structures des classes java *Produit* et *Stat* héritant toutes les deux de *Object* .

```
MyCsvUtil.writeValuesAsCsvFile(listeProduits,"produits.csv");
```

génère automatiquement

```
id;label;prixHt;tauxTva  
1;stylo bille rouge;1.6;20  
2;grand classeur;4.6;20  
3;produitXyz;4.4;10  
4;gomme;2.1;20
```

```
MyCsvUtil.writeValuesAsCsvFile(listeStats,"stats.csv");
```

génère automatiquement

```
label;somme;moyenne;ecartType  
statHt;12.7;3.175;1.3386093530227552  
statTtc;14.8;3.7;1.5143315356948757
```

### 1.3. Introspection avec annotations

L'annexe sur les annotations permettra de mieux comprendre l'exemple ci-dessous.

#### CsvIgnore.java

```
package util;
import java.lang.annotation.Documented; import java.lang.annotation.ElementType;
import java.lang.annotation.Retention; import java.lang.annotation.RetentionPolicy; import java.lang.annotation.Target;

@Documented
@Target(ElementType.FIELD) //utilisable au dessus d'un field/attribut
@Retention(RetentionPolicy.RUNTIME) //conservee dans code compile et accessible au runtime
public @interface CsvIgnore {
    //...
}
```

```
...
import com.fasterxml.jackson.annotation.JsonIgnore;
import util.CsvIgnore;
```

```
public class Stat {
    private String label;
    private Double somme;
    private Double moyenne;

    @CsvIgnore
    @JsonIgnore
    private Double ecartType;
    ...
}
```

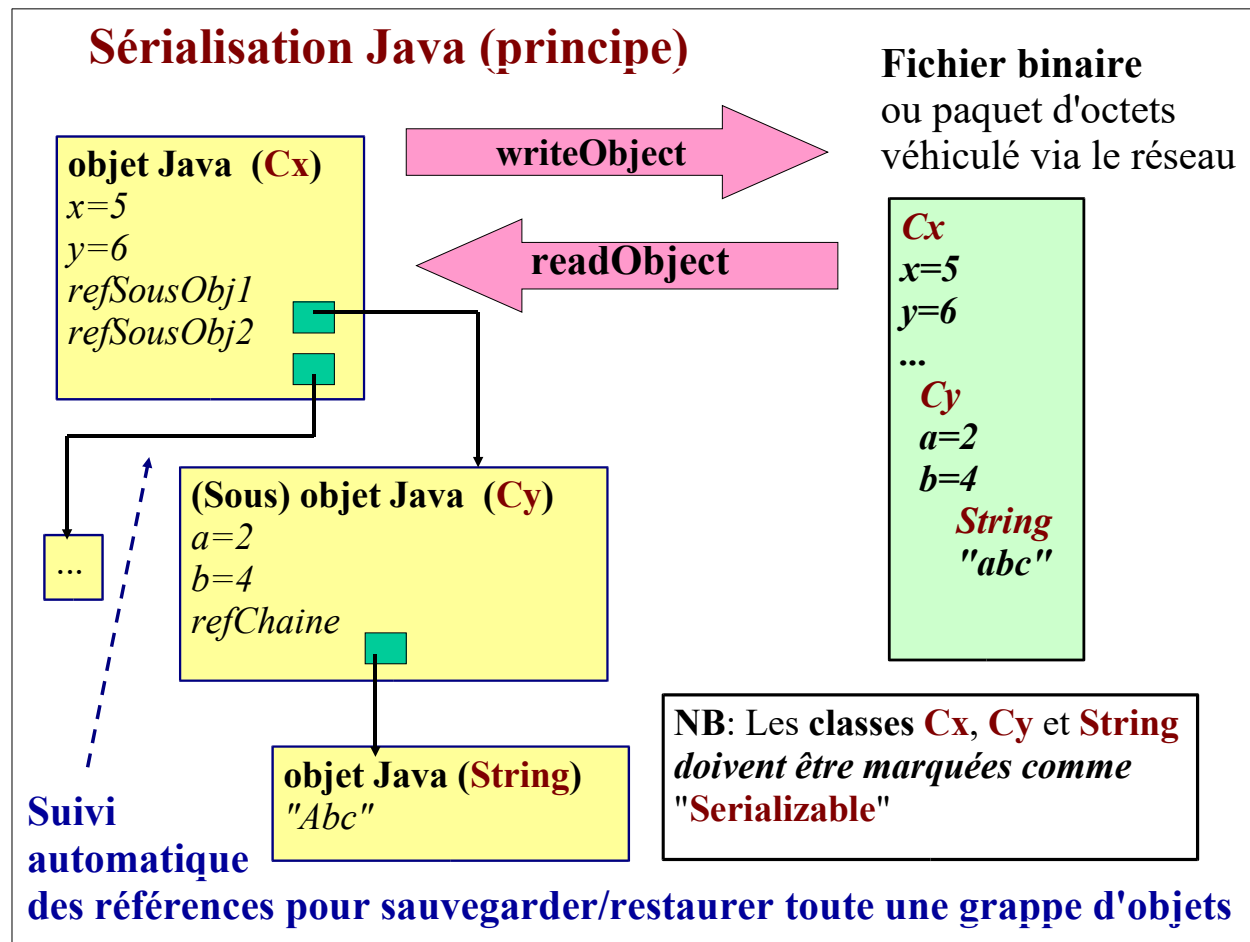
```
...
Field[] tabChamps = c.getDeclaredFields();
for(Field f : tabChamps) {
    CsvIgnore annotCsvIgnore = f.getAnnotation(CsvIgnore.class);
    //si annotCsvIgnore==null , @CsvIgnore n'a pas été placé au dessus de f
    if(annotCsvIgnore==null) {
        String fieldName= f.getName();
        csvString=(csvString==null)?fieldName:csvString + ";" +fieldName;
    }
}
....
```

Fichier **stats.csv** généré en ignorant la partie ecartType :

```
label;somme;moyenne
statHt;12.7;3.175
statTtc;14.8;3.7
```



## 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:

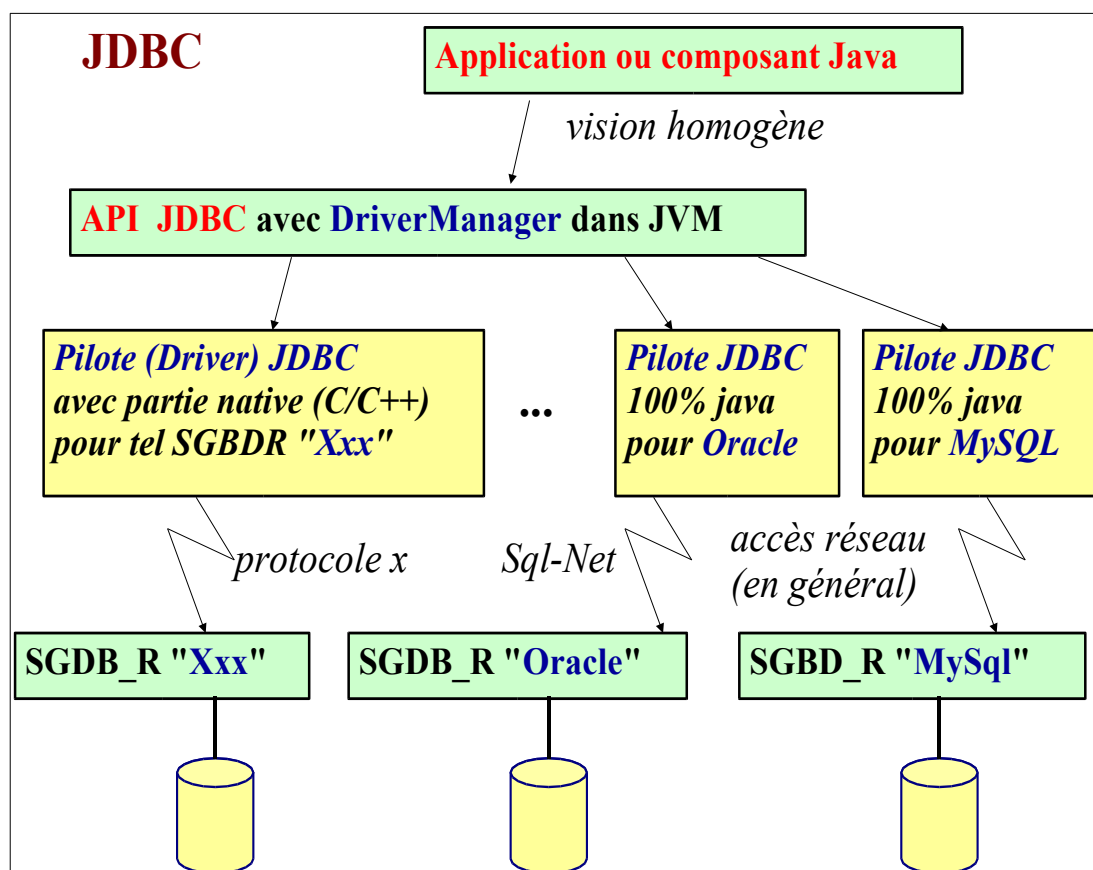
- L'API "**JAXB**" (*Java Api for Xml Binding*) permet de gérer une persistance au format XML
- L'API "**JPA/Hibernate**" permet de gérer une persistance au sein d'une base de données relationnelle (MySQL , Oracle , ...)
- L'API "**Jackson-databind**" permet de gérer une persistance au format JSON

# 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
  - \* ...





## 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("com.mysql.cj.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_2="jdbc:mysql://localhost/test"; // version MySql

// Ouverture de la connexion:
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 , necessite mysql-connector-...jar ou autre dans le CLASSPATH
driver=com.mysql.cj.jdbc.Driver
#url de la base de donnée
url=jdbc:mysql://localhost:3306/mydb?serverTimezone=UTC
username=root
password=root
```

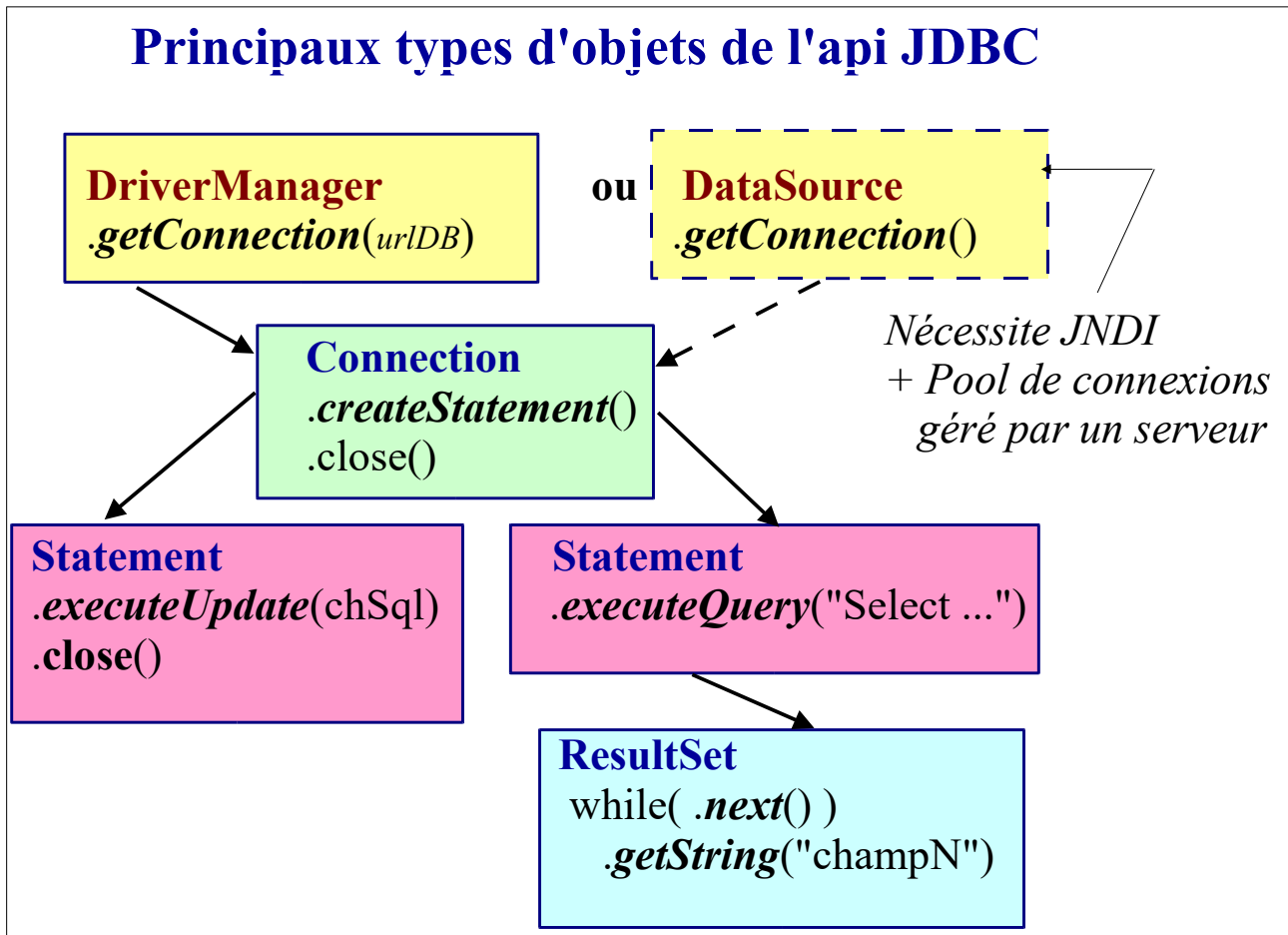
```
ResourceBundle ressources = ResourceBundle.getBundle("paramDB") ; // paramDB.properties
String driver = ressources.getString("driver"); String chUrl = ressources.getString("url");
String username = ressources.getString("username"); String password = ressources.getString("password");
Class.forName(driver); 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 mäj.
else
    connexion.rollback(); // pour annuler toutes les mäj 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

```

/*Integer*/ Long recupValeurAutoIncrPk(PreparedStatement pst){
    /*Integer*/ Long pk=null;
    try {
        ResultSet rsKeys = pst.getGeneratedKeys();
        if(rsKeys.next()){ pk= rsKeys.getLong(1); /*rsKeys.getInt(1);*/ }
    } catch (SQLException e) { e.printStackTrace(); }
    return pk;
}

public Long insertNewCompte(Compte cpt){
    Long pk=null; Connection cn = initConnection();
    PreparedStatement pst = null;
    String reqSql = "insert into Compte(label,solde) values(?,?)" ;
    try { pst = cn.prepareStatement( reqSql, Statement.RETURN_GENERATED_KEYS );
        pst.setString(1, cpt.getLabel());  pst.setDouble(2, cpt.getSolde());
        pst.executeUpdate(); //avec auto_increment mysql sur colonne numCpt
        pk = recupValeurAutoIncrPk (pst) ;
    } 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 **mise à jour** depuis l'objet **ResultSet** (méthodes UpdateXXX() )
- ...

### 13.1. Types de champs

La classe `java.sql.Types` comporte les constantes suivantes (types **XOPEN**) :

<b>ARRAY</b>	SQL ARRAY (Depuis JDBC2)
<b>BIGINT</b>	
<b>BINARY</b>	
<b>BIT</b>	
<b>BLOB</b>	Binary Large Object (ex: image) (Depuis JDBC2)
<b>CHAR</b>	
<b>CLOB</b>	Character Large Object (ex: memo) (Depuis JDBC2)
<b>DATE</b>	
<b>DECIMAL</b>	
<b>DISTINCT</b>	User Defined Data Type (Depuis JDBC2)
<b>DOUBLE</b>	
<b>FLOAT</b>	
<b>INTEGER</b>	
<b>JAVA_OBJECT</b>	Objet Java (Depuis JDBC2)
<b>LONGVARIABLE</b>	
<b>LONGVARCHAR</b>	
<b>NULL</b>	
<b>NUMERIC</b>	
<b>OTHER</b>	
<b>REAL</b>	
<b>REF</b>	(Depuis JDBC2)
<b>SMALLINT</b>	
<b>STRUCT</b>	User Defined Data Type (Depuis JDBC2)
<b>TIME</b>	
<b>TIMESTAMP</b>	
<b>TINYINT</b>	
<b>VARIABLE</b>	
<b>VARCHAR</b>	

==> 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 a 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

<b>SELECT</b> fieldlist	<b>FROM</b> tablenames	[ <b>WHERE</b> searchcondition]
[ <b>GROUP BY</b> fieldlist	[ <b>HAVING</b> searchconditions] ]	[ <b>ORDER BY</b> 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 - Threads (java)

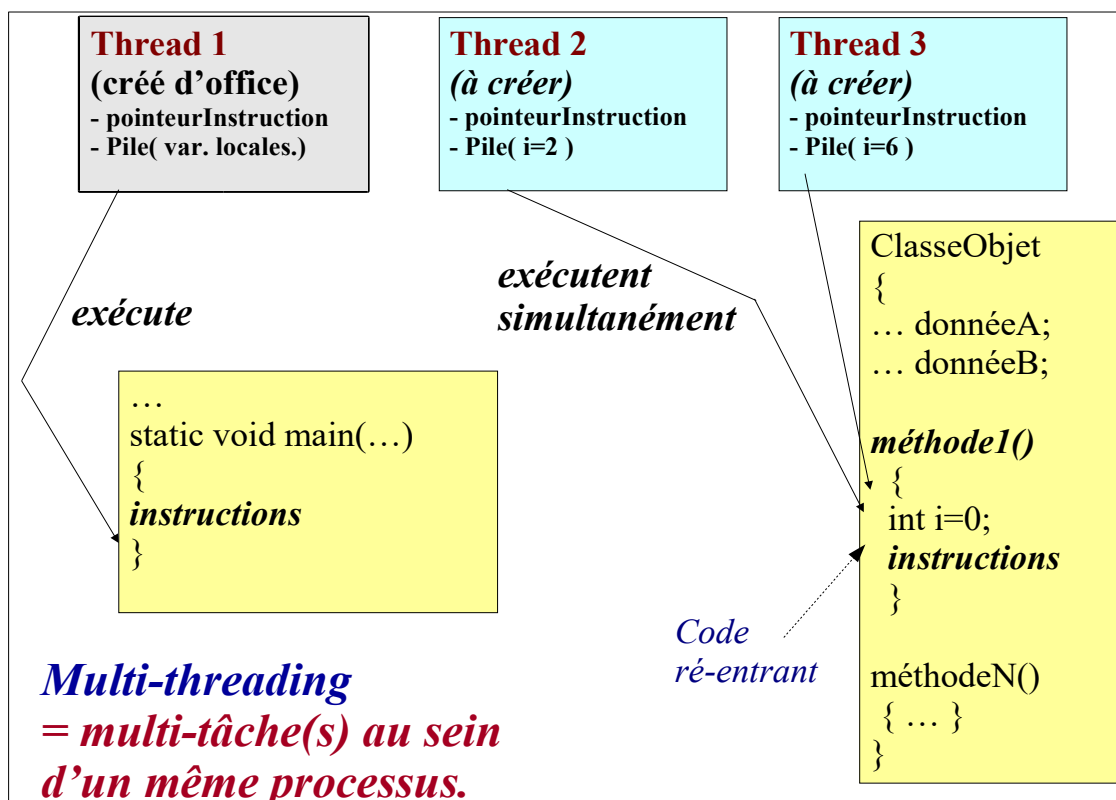
## 1. Concept de threads

### 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.

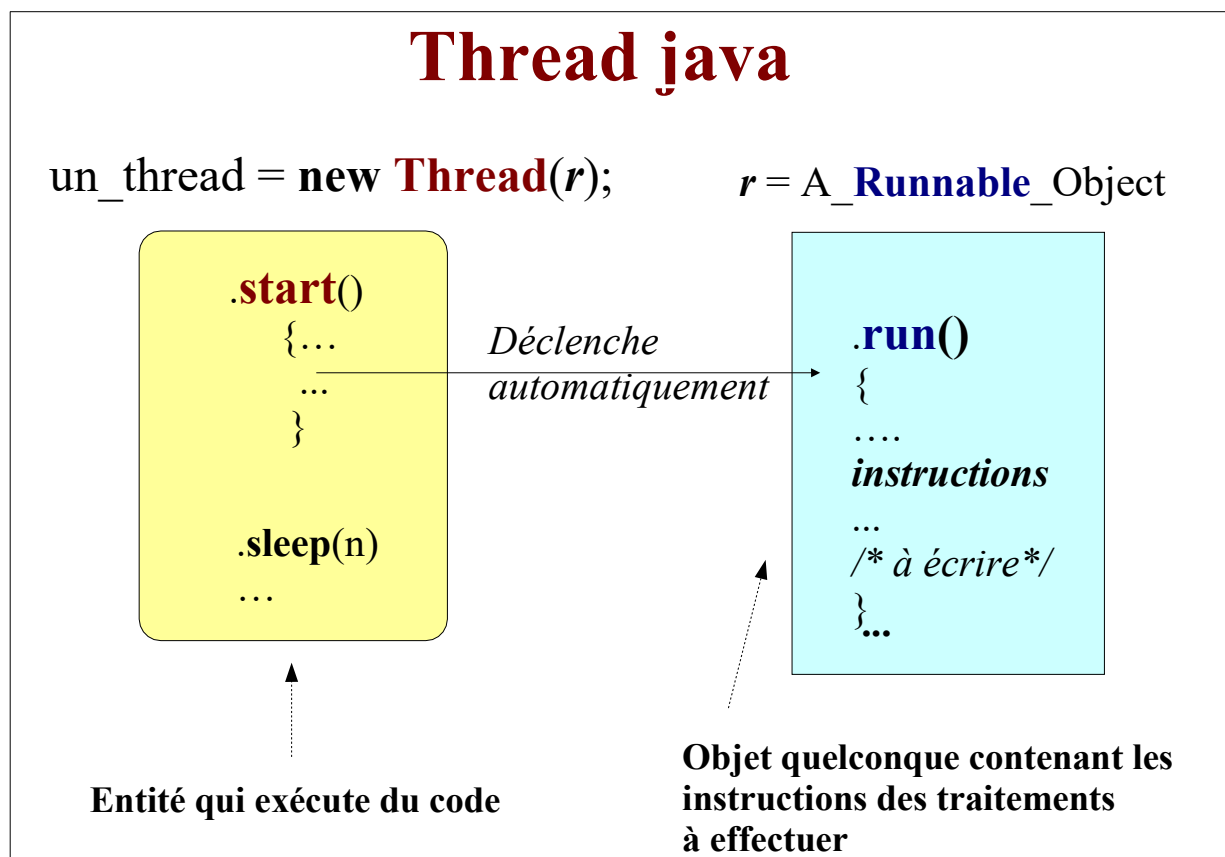




Remarque: Il existe toujours au moins un thread au sein d'une application ou applet JAVA. Ce thread principal est créé automatiquement, il s'occupe de la gestion des événements associés à l'interface graphique ou bien de l'enchaînement des instructions partant de la méthode statique `main()` en mode 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
    }

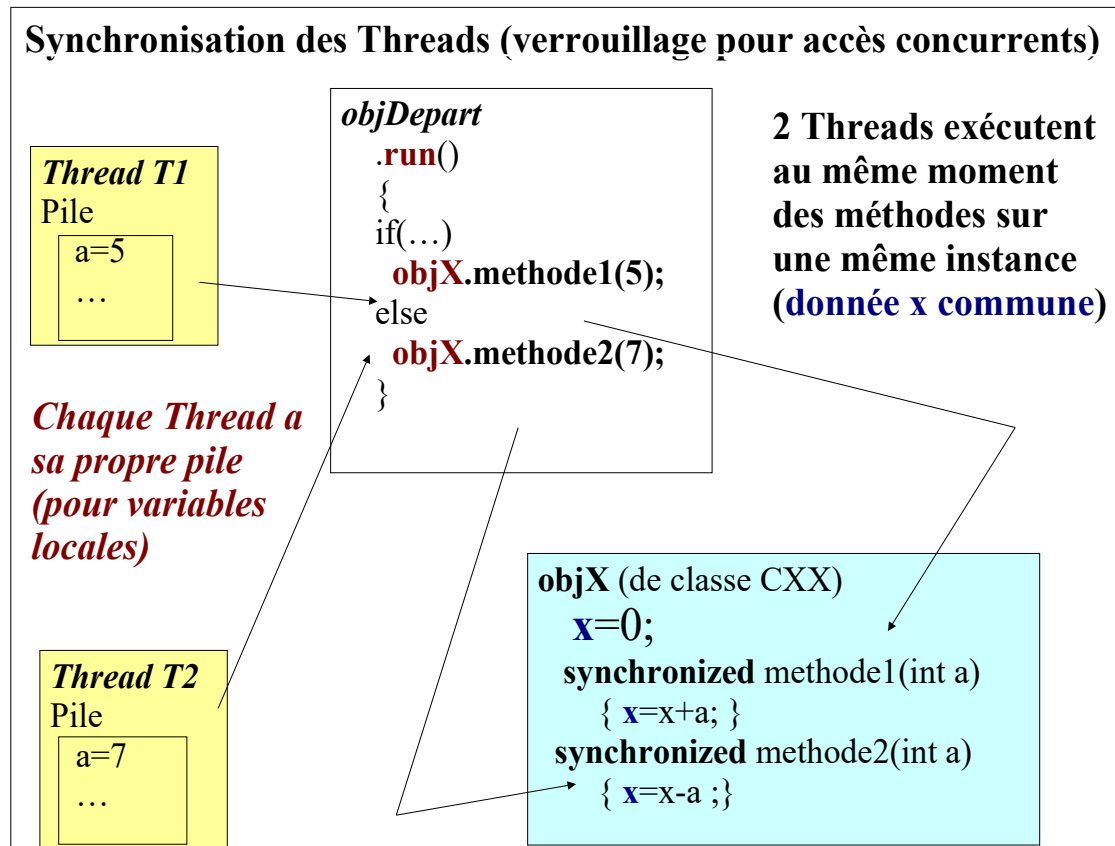
    private void arreter_thread()
    { fin=true; }

    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 .

Lorsque 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 ) java 1.0

### 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()** .

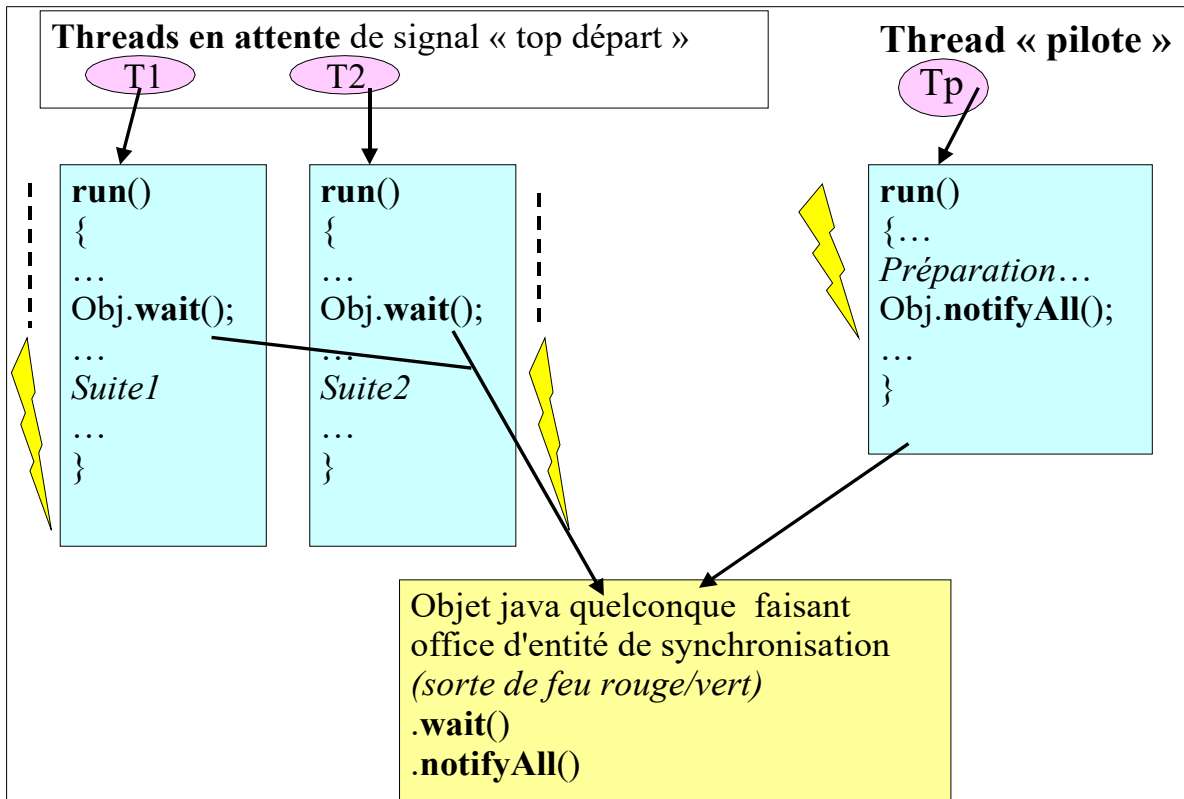
NB: Les méthodes **wait()** et **notify()** ne peuvent être appelées sur une instance que si elles sont placées dans un bloc "**synchronized**" .

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 qu'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

## 4.3. 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* ).

# 5. Concurrent api

## 5.1. interface Runnable (java.lang)

Depuis java 8 , l'interface fondamentale Runnable est devenue une interface (uni-)fonctionnelle.

```
public interface Runnable {
    public void run();
}
```

```
Runnable r = () -> { /*code de la tâche*/ }
```

## 5.2. interface Callable<T> (java.util.concurrent )

```
public interface Callable<T> {
    public T call();
}
```

<code>Callable&lt;T&gt; c = () -&gt; { /*corps de la tâche*/ <b>return</b> t; }</code>
--

Callable<T> existe depuis le jdk 1.5 et est depuis java 1.8 vue comme une interface (uni-)fonctionnelle .

### 5.3. interface Future<T> (java.util.concurrent)

Disponible depuis le jdk 1.5, un objet **Future<T>** en **java** correspond partiellement à la notion de **Promise** en *javascript* .

Un objet technique de ce type (*recupéré immédiatement lors d'un lancement de traitement long*) **permettra de récupérer un résultat en différé** (dans le futur) .

boolean <b>isDone()</b>	teste la terminaison du thread pour savoir si la donnée résultante de son exécution est disponible. Retourne true si ce thread s'est bien terminé ou s'il a malheureusement levé une exception pendant son exécution, ou enfin s'il a été suspendu.
T <b>get()</b>	retourne le résultat (instance de T) de la tâche exécutée par le thread. Si le thread n'a pas terminé son exécution, alors l'appel de cette méthode est bloqué en attente active jusqu'à ce qu'il termine.
boolean <b>cancel(true)</b>	Demande à arrêter la tâche si elle n'est pas finie , le paramètre d'entrée "mayInterruptIfRunning" est souvent fixé à true . La valeur de retour (souvent ignorée) est à true si tâche interrompue ou false si tâche déjà terminée .
boolean <b>isCancelled()</b>	renvoi true si tâche "interrompue" avant la fin .
T <b>get(long timeout, TimeUnit unit)</b>	variante avec <b>timeout</b> de la méthode <b>.get()</b> <b>Si par exemple après un timeout de 1500 TimeUnit.MILLISECONDS</b> la tâche n'est toujours pas finie , cette méthode remonte une exception de type <b>TimeoutException</b> à rattraper via un try/catch pour nous signaler que cette tâche n'est pas terminée .



## 5.4. ExecutorService

Au coeur du package `java.util.concurrent` l'interface **ExecutorService** comporte les principales méthodes suivantes :

<b><code>void execute(Runnable command)</code></b>	lance l'exécution (via un thread créé ou disponible) d'une tâche de type <code>Runnable</code> , ne retourne rien.
<b><code>Future&lt;T&gt; submit(Callable&lt;T&gt; task)</code></b>	lance l'exécution d'une tâche de type <code>Callable&lt;T&gt;</code> , retourne un objet de type <code>Future&lt;T&gt;</code> permettant de récupérer ultérieurement (en différé) un résultat de type <code>T</code> .
<b><code>Future&lt;?&gt; submit(Runnable task)</code></b>	Comme <code>execute()</code> mais retournant <code>Future&lt;?&gt;</code> pour attente du résultat ou de la fin
<b><code>T invokeAny(Collection&lt;? extends Callable&lt;T&gt;&gt; tasks)</code></b> ou bien <b><code>T invokeAny(... tasks, long timeout, ...)</code></b>	Lance (via des threads ) une liste de tâches et attend <u>en mode bloquant</u> la première réponse , les autres tâches moins rapides sont automatiquement annulées/stoppées.
<b><code>List&lt;Future&lt;T&gt;&gt; invokeAll(Collection&lt;? extends Callable&lt;T&gt;&gt; tasks)</code></b> <i>existe en version avec timeout</i>	lance <u>en mode bloquant</u> plein de tâches et récupère une liste de résultats encapsulés dans des <code>Future&lt;T&gt;</code> quand tout est prêt/fini .
Autres méthodes	<code>shutdown()</code> , <code>awaitTermination(timeout,...)</code> , ...

Une instance d'une classe implémentant l'interface `ExecutorService` pourra être créée via

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

ou bien

```
ExecutorService executor = Executors.newFixedThreadPool(3/*nThreads*/)
```

ou bien

```
ExecutorService executor = Executors.newCachedThreadPool();
```

ou bien d'autres façon encore .

<b><i>newSingleThreadExecutor()</i> ;</b>	Un seul nouveau thread pouvant lancer plusieurs tâches alors exécutées séquentiellement les unes après les autres .
<b><i>newFixedThreadPool(3/*nThreads*/)</i></b>	via un pool de threads (en //) de taille maxi à paramétrer (si plus de tâches à exécuter que de threads dispos --> attente automatique via queue) . chaque thread ne sera arrêté que si appel explicite à <code>.shutdown()</code> .
<b><i>newCachedThreadPool()</i>;</b>	via un pool de threads dont la taille est automatiquement ajustée en fonction des besoins (à la hausse ou à la baisse si rien à faire durant 60s )
autres	voir javadoc Executors

## 5.5. Exemple "EssaiExecutors"

### MyRunnableCode.java

```
package tp.langage.thread;

public class MyRunnableCode implements Runnable {
    private String prefix;

    public MyRunnableCode() {super(); this.prefix = ""; }
    public MyRunnableCode(String prefix) { super(); this.prefix = prefix; }

    @Override
    public void run() {
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {}
        System.out.println(prefix + Thread.currentThread().getName());
    }
}
```

### EssaiExecutors.java

```
package tp.langage.thread;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

//ExecutorService depuis jdk 1.5
public class EssaiExecutors {

    // Un "ExecutorService" (à fabriquer via Executors.new....Executor()) démarre automatiquement des
    // Threads (rangés dans des "pools") pour exécuter des instances de Callable<T> ou de Runnable
    public static void main(String[] args) {

        MyRunnableCode myRunnableCode = new MyRunnableCode("");

        ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();
                                                    //exécution séquentielle en background

        singleThreadExecutor.submit(myRunnableCode);
        singleThreadExecutor.submit(myRunnableCode);
        singleThreadExecutor.submit(myRunnableCode);
        singleThreadExecutor.shutdown();//automatiquement différé
    }
}
```

```

MyRunnableCode myRunnableCode2 = new MyRunnableCode("#");

ExecutorService multiThreadExecutor = Executors.newFixedThreadPool(3);
//exécutions multiples (en //) en background
multiThreadExecutor.submit(myRunnableCode2);
multiThreadExecutor.submit(myRunnableCode2);
multiThreadExecutor.submit(myRunnableCode2);
multiThreadExecutor.shutdown();//automatiquement différé , .shutdownNow() pour arrêt immédiat

MyRunnableCode myRunnableCode3 = new MyRunnableCode("@");

ScheduledExecutorService scheduleExecutor =
    Executors.newSingleThreadScheduledExecutor();
System.out.println("Lancement d'un thread/tâche (@) en différé (2000ms)");
scheduleExecutor.schedule(myRunnableCode3, 2000, TimeUnit.MILLISECONDS);
scheduleExecutor.shutdown();//automatiquement différé , .shutdownNow() pour arrêt immédiat
}
}

```

#### Résultats :

```

Lancement d'un thread (@) en différé (2000ms)
#pool-2-thread-3
#pool-2-thread-2
*pool-1-thread-1
#pool-2-thread-1
*pool-1-thread-1
*pool-1-thread-1
@pool-3-thread-1

```

## 5.6. Exemple "TestFuture" (avec Executors et Callable<T>)

### LongTask.java

```
package tp.langage.thread;

public class LongTask {

    public static void printThread() {
        System.out.println(Thread.currentThread().getName());
    }

    public static void simulateLongTask(String msg, long nbMs) {
        try {
            System.out.println(">>(begin)" + msg + " / by " + Thread.currentThread().getName());
            Thread.sleep(nbMs);
            System.out.println("<<(end)" + msg + " / by " + Thread.currentThread().getName());
        } catch (InterruptedException e) {
            //e.printStackTrace();
            System.out.println("*** interrupted ***");
        }
    }
}
```

### CallableComputing.java

```
package tp.langage.thread;
import java.util.concurrent.Callable;

public class CallableComputing implements Callable<String> {
    private double x;

    @Override
    public String call() throws Exception {
        LongTask.simulateLongTask("long computing task (in background) ...", 5000);
        return String.valueOf(Math.sqrt(x));
    }

    public CallableComputing() {super(); this.x = 0; }
    public CallableComputing(double x) {super(); this.x = x; }

    public double getX() { return x; }
    public void setX(double x) {this.x = x; }
}
```

**EssaiFuture.java**

```

package tp.langage.thread;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class EssaisFuture {

    public static void main(String[] args) {
        //NB: Callable<T>/call() ressemble un peu à l'interface Running/run()
        //mais permet de récupérer (ultérieurement) un résultat via Future<T> .
        Callable<String> c = new CallableComputing(9);
        String result=null;

        ExecutorService executor = Executors.newSingleThreadExecutor();

        Future<String> futureRes = executor.submit(c);
        while(result==null){
            LongTask.simulateLongTask("other works ...",2000);
            if(futureRes.isDone()){
                try {
                    result = futureRes.get();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (ExecutionException e) {
                    e.printStackTrace();
                }
            }
        }
        System.out.println("result=" + result);

        System.out.println("-----");
        Future<String> futureRes2 = executor.submit(c);
        LongTask.simulateLongTask("other works ...",2000);
        if(futureRes2.isDone()){
            try {
                result = futureRes2.get();
                System.out.println("result2=" + result);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (ExecutionException e) {
                e.printStackTrace();
            }
        }
        else {

```

```

    futureRes2.cancel(true);
    if(futureRes2.isCancelled()){
        System.out.println("background computing was cancelled");
    }
}
System.out.println("-----");
Future<String> futureRes3 = executor.submit(c);
LongTask.simulateLongTask("other works ...",2000);
try {
    result = futureRes3.get(1500,TimeUnit.MILLISECONDS);
    System.out.println("result3=" + result);
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
} catch (TimeoutException e) {
    System.err.println("tâche 3 toujours pas terminée au bout de 1500ms");
    System.out.println("after 1500ms,futureRes3.isDone()="+futureRes3.isDone());
    System.out.println("after 1500ms,futureRes3.isCancelled()="+futureRes3.isCancelled());
    //e.printStackTrace(); }
}
    executor.shutdown();//automatiquement différé , .shutdownNow() pour arrêt immédiat
}

```

Résultats :

```

>>(begin)other works ... / by main
>>(begin)long computing task (in background) ... / by pool-1-thread-1
<<(end)other works ... / by main
>>(begin)other works ... / by main
<<(end)other works ... / by main
>>(begin)other works ... / by main
<<(end)long computing task (in background) ... / by pool-1-thread-1
<<(end)other works ... / by main
result=3.0
-----
>>(begin)other works ... / by main
>>(begin)long computing task (in background) ... / by pool-1-thread-1
<<(end)other works ... / by main
** interrupted **
background computing was cancelled
-----
>>(begin)other works ... / by main
>>(begin)long computing task (in background) ... / by pool-1-thread-1
<<(end)other works ... / by main
after 1500ms,futureRes3.isDone()==false
after 1500ms,futureRes3.isCancelled()==false
tâche 3 toujours pas terminée au bout de 1500ms
<<(end)long computing task (in background) ... / by pool-1-thread-1

```

## 5.7. Semaphore (Synchronisation de Threads)

Un **Semaphore** est (en java depuis le jdk 1.5) un objet technique de synchronisation entre différents threads .

Un **sémaphore** correspond conceptuellement à un **ensemble de jetons disponible** .

Un thread doit **acquérir un jeton disponible** (via un **appel bloquant** à semaphore.acquire() ou bien semaphore.tryAcquire(timeout...) *pour pouvoir ensuite travailler seul sur une ressource partagée* .

En appelant la méthode symétrique semaphore.release() , un thread peut rendre un jeton dans le semaphore (souvent après avoir terminé un certain travail en mode "exclusivité" ) . Cette action va immédiatement débloquer d'éventuelles attentes exprimées via semaphore.acquire() .

Plus précisément, un **sémaphore** encapsule un entier, avec une contrainte de positivité, et deux opérations atomiques d'incrément et de décrémentation :

- via le constructeur : variable entière (toujours positive ou nulle) ;
- opération -- (**acquire()**) : décrémente le compteur s'il est strictement positif ; bloque s'il est nul en attendant de pouvoir le décrémentation ;
- opération ++ (**release()**) : incrémente le compteur.

**NB :** Depuis java 1.5 , les **Semaphores** constituent un mécanisme de synchronisation **plus souple et plus fiable** que le mécanisme .wait()/notify() disponible dès java 1.0 sur la classe Object .

### EssaiSemaphore.java

```
package tp.langage.thread;
import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

public class EssaiSemaphore {
    public static void main(String[] args) {
        Semaphore semaphore = new Semaphore(0);

        Thread t = new Thread(()-> { LongTask.simulateLongTask("background thread work ...", 3000);
                                semaphore.release()});

        t.start();
        System.out.println("... faire autre chose ...");

        //attendre la disponibilité du sémaphore:
        try {
            //semaphore.acquire();
            if(semaphore.tryAcquire(5, TimeUnit.MINUTES)){
                System.out.println("semaphore acquis");
            }else{
                System.out.println("after 5mn (semaphore toujours pas disponible)");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

}

Résultats :

```

... faire autre chose ...
>>(begin)background thread work ... / by Thread-0
<<(end)background thread work ... / by Thread-0
semaphore acquis

```

La notion de "**Mutex**" (mutuelle exclusion) peut s'implémenter en java avec **un sémaphore à un seul jeton** .

## 5.8. CompletableFuture (attendre et consommer résultats produits)

### EssaiCompletableFuture.java

```

package tp.langage.thread;

import java.util.concurrent.Callable;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

//CompletableFuture depuis jdk1.6 avec méthode take() retournant un Future
//logique producteur/consommateur

public class EssaiCompletableFuture {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(4); //jusqu'à 4 threads en //
        CompletableFuture<String> completableFuture =
            new ExecutorCompletionService<String>(executor);

        double[] tabVal = { 4, 9 , 16 , 25, 36, 49 , 64, 81, 100 };
        int taille = tabVal.length;
        for(int i=0;i<taille;i++){
            Callable<String> c = new CallableComputing(tabVal[i]);
            //CompletableFuture<String> encapsule l'executor et est typé comme Future<T>
            completableFuture.submit(c); //lancement asynchrone d'un "producteur"
        }
        for(int i=0;i<taille;i++){
            try {
                Future<String> futureRes = completableFuture.take(); //attente du PREMIER TERMINE
                System.out.println(futureRes.get()); //consommateur simple
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}

```



```

    }
    }
    System.out.println("fin-main");
    //arrêter l'executor (si besoin en différé) :
    executor.shutdown();
}
}

```

Résultats :

```

>>(begin)long computing task (in background) ... / by pool-1-thread-2
>>(begin)long computing task (in background) ... / by pool-1-thread-3
>>(begin)long computing task (in background) ... / by pool-1-thread-4
>>(begin)long computing task (in background) ... / by pool-1-thread-1
<<(end)long computing task (in background) ... / by pool-1-thread-2
<<(end)long computing task (in background) ... / by pool-1-thread-4
<<(end)long computing task (in background) ... / by pool-1-thread-3
<<(end)long computing task (in background) ... / by pool-1-thread-1
>>(begin)long computing task (in background) ... / by pool-1-thread-3
3.0
4.0
2.0
>>(begin)long computing task (in background) ... / by pool-1-thread-1
>>(begin)long computing task (in background) ... / by pool-1-thread-2
>>(begin)long computing task (in background) ... / by pool-1-thread-4
5.0
<<(end)long computing task (in background) ... / by pool-1-thread-3
<<(end)long computing task (in background) ... / by pool-1-thread-1
<<(end)long computing task (in background) ... / by pool-1-thread-2
>>(begin)long computing task (in background) ... / by pool-1-thread-1
6.0
<<(end)long computing task (in background) ... / by pool-1-thread-4
8.0
7.0
9.0
<<(end)long computing task (in background) ... / by pool-1-thread-1
10.0
fin-main

```

## 6. ForkJoin

### 6.1. basic fork/join

```
public class EssaiBasicForkJoin {
    public static void main(String[] args) {
        System.out.println("debut - main");
        Thread t = new Thread(new MyRunnableCode());
        t.start(); // basic sort of fork()
        System.out.println(("suite - main / avant join"));
        try {
            t.join(); //attente de la fin de l'exécution du thread
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("fin main - apres join");
    }
}
```

-->

```
debut - main
suite - main / avant join
Thread-0
fin main - apres join
```

### 6.2. fork/join (depuis java 1.7)

Le framework **fork / join** en Java (depuis le jdk 1.7) est idéal pour **un problème qui peut être divisé en parties plus petites et résolu en parallèle**.

Les étapes fondamentales d'un problème fork / join sont les suivantes:

- **Diviser le problème en plusieurs morceaux**
- **Résoudre chacune des pièces en parallèle**
- **Combinez chacune des sous-solutions en une solution globale**

Une [ForkJoinTask](#) est l'interface qui définit un tel problème. On s'attend généralement à ce que vous sous-classiez l'une de ses implémentations abstraites (généralement la [RecursiveTask](#)) plutôt que d'implémenter l'interface directement.

Détail technique :

Le *ForkJoinPool* est le coeur du framework. Il s'agit d'une implémentation de [ExecutorService](#) qui gère les threads de travail et nous fournit des outils pour obtenir des informations sur l'état et les performances du pool de threads.

Les threads de travail ne peuvent exécuter qu'une tâche à la fois, mais *ForkJoinPool* ne crée pas de thread séparé pour chaque sous-tâche. Au lieu de cela, chaque thread du pool a sa propre file d'attente à deux extrémités (double ended *queue[deque]*) qui stocke les tâches.

Cette architecture est essentielle pour équilibrer la charge de travail du thread à l'aide de l'algorithme "work-stealing".

## 6.3. fork/join appliqué sur un quickSort en mode "multi-processeurs"

### MyQuickSortAlgo.java (version sans fork/join)

```
package tp.langage.thread;
public class MyQuickSortAlgo {

    static void echanger(double[] tableau ,int indice1 ,int indice2){
        double temp = tableau[indice1];
        tableau[indice1] = tableau[indice2];
        tableau[indice2] = temp;
    }

    static int partition(double[] tableau,int deb,int fin){
        int indicePivot=deb; //au sens indice initial qui va évoluer
        double valeurPivot=tableau[deb]; //valeur du pivot (= arbitrairement valeur en première position du tableau)
        //via une future permutation , cette valeur sera à une future autre position

        for(int i=deb+1;i<=fin;i++){
            if (tableau[i]<valeurPivot){
                indicePivot++; //nouvelle valeur pour le futur indice du pivot (qui peut encore évoluer selon boucle en cours)
                echanger(tableau,indicePivot,i); //pour placer à la "future gauche" de l'indice provisoire du pivot
                //tous les éléments plus petits que le pivot
            }
        }
        echanger(tableau,deb,indicePivot); //permutation pour que la valeur du pivot soit rangée à sa place (précédemment calculée)
        //et pour qu'un des éléments plus petits soit placé au début (à gauche )

        return indicePivot;
    }

    //version ordinaire (sans optimisation multi-proc):
    static void tri_rapide(double[] tableau,int deb,int fin){
        if(deb<fin){
            //partitionner le tableau en 2 parties partiellement ré-arrangées .
            //d'un coté tous les éléments plus petits que le pivot , de l'autre coté tous les éléments plus grands:
            int positionPivot=partition(tableau,deb,fin);
            tri_rapide(tableau,deb,positionPivot-1); //trier le sous tableau des plus petits éléments que le pivot
            tri_rapide(tableau,positionPivot+1,fin); //trier le sous tableau des plus grands éléments que le pivot
        }
    }

    static void quick_sort(double[] tableau){
        tri_rapide(tableau, 0, tableau.length - 1 );
    }
}
```

### MyQuickSortMultiProc.java (avec fork/join)

```
package tp.langage.thread;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;
```

*//ForkJoin (RecursiveAction) / divide & conquer since jdk 1.7*

```
public class MyQuickSortMultiProc extends RecursiveAction {

    private static final long serialVersionUID = 1L;
    private static final int FORK_JOIN_MIN_SIZE=1024;

    private double[] tab;
    private int start,end;

    public MyQuickSortMultiProc(double[] tableau,int deb,int fin){
        this.tab = tableau;
        this.start = deb;
        this.end = fin;
    }

    static void quick_sort_multiProc(double[] tableau){
        MyQuickSortMultiProc myQuickSortMultiProc=
            new MyQuickSortMultiProc(tableau, 0, tableau.length - 1);
        ForkJoinPool threadPool = new ForkJoinPool();
        threadPool.invoke(myQuickSortMultiProc);
    }

    @Override //RecursiveAction
    protected void compute() {
        // pas de paramètre et donc les tab et indices
        // doivent être renseignés en tant qu'attributs + constructeurs
        MyQuickSortMultiProc sousTriGaucheViaForkJoin=null;
        MyQuickSortMultiProc sousTriDroitViaForkJoin = null;
        //System.out.println("MyQuickSortMultiProc.compute() executé par "+Thread.currentThread().getName() );

        if(start<end){
            //partitionner le tableau en 2 parties partiellement ré-arrangées .
            //d'un coté tous les éléments plus petits que le pivot , de l'autre coté tous les éléments plus grands:
            int positionPivot=MyQuickSortAlgo.partition(tab,start,end);

            //NB: étant donné que la version forkJoin ajoute une complexité au niveau du code
            // (instance à créer , thread à gérer) , cette version ne sera activée/utilisée
            //que pour trier des sous tableaux dont la taille minimum est supérieure à
            //FORK_JOIN_MIN_SIZE=1024

            if(positionPivot - start > FORK_JOIN_MIN_SIZE){
                sousTriGaucheViaForkJoin= new MyQuickSortMultiProc(tab, start, positionPivot-1);
                sousTriGaucheViaForkJoin.fork(); //déléguer le tri du sous tableau des plus petits éléments que le pivot
            } else MyQuickSortAlgo.tri_rapide(tab, start, positionPivot-1);

            if(end - positionPivot > FORK_JOIN_MIN_SIZE){
                sousTriDroitViaForkJoin= new MyQuickSortMultiProc(tab, positionPivot+1,end);
                sousTriDroitViaForkJoin.fork(); //déléguer le tri du tableau des plus grands éléments que le pivot
            } else MyQuickSortAlgo.tri_rapide(tab, positionPivot+1, end);
        }
    }
}
```

```

if(sousTriGaucheViaForkJoin!=null) sousTriGaucheViaForkJoin.join(); //attendre
if(sousTriDroitViaForkJoin!=null) sousTriDroitViaForkJoin.join(); //attendre

//NB: il existe invokeAll(recursiveAction1 , recursiveAction2) qui declenche en // .fork() et .join()
    }
}
}

```

### TestMultiProcesseurs.java (tests avec rapidités comparées)

```

package tp.langage.thread;

public class TestMultiProcesseurs {
    public static void main(String[] args) {
        System.out.println("nb processors:" + Runtime.getRuntime().availableProcessors());
        double[] t1 = produce_init_tab();
        double[] copyOfT1 = t1.clone();
        display_tab(t1); //display_tab(copyOfT1);
        System.out.println("tri ordinaire (quick-sort) ");
        test_tri(t1);
        System.out.println("tri (quick-sort) optimisé pour machine multi-processeurs ");
        test_tri_multiProc(copyOfT1);
    }

    static double[] produce_init_tab() {
        //double[] t = { 5,2,1,9,3,4,12,8,16,6 };
        //double[] t = { 26,7,5,2,1,9,3,4,34,12,8,16,6,78,10,89,33,23,90,123,72,3,48 };
        //final int taille=10;
        final int taille=1024*1024*8;
        double[] t = new double[taille];
        for(int i=0;i<taille;i++){
            t[i]=Math.random()*taille;
        }
        return t;
    }

    static void display_tab(double[] tab){

```

```

        if(tab.length <= 30) {
            for(double x : tab)
                System.out.print(x + " ");
            System.out.print("\n");
        } else{ System.out.println("tableau de taille = " + tab.length);
        }
    }

    static void test_tri(double[] tab){
        long td = System.nanoTime();
        MyQuickSortAlgo.quick_sort(tab);
        long tf = System.nanoTime();    display_tab(tab);
        System.out.println("## " + (tf-td)/ 1000000 + " ms");
    }

    static void test_tri_multiProc(double[] tab){
        long td = System.nanoTime();
        MyQuickSortMultiProc.quick_sort_multiProc(tab);
        long tf = System.nanoTime();    display_tab(tab);
        System.out.println("** " + (tf-td) / 1000000 + " ms");
    }
}

```

#### Résultats (avec i7):

```

nb processors:4
tableau de taille = 8388608
## tri ordinaire (quick-sort)
tableau de taille = 8388608
## 872 ms
** tri (quick-sort) optimisé pour machine multi-processeurs
tableau de taille = 8388608
** 538 ms

```

**Autre exemple de fork/join (calcul de moyenne ou d'écartType).*****SequentialComputing.java***

```

package tp.thread.sam;
//interface d'une référence de fonction pour calcul ordinaire de somme ou moyenne ou ...
public interface SequentialComputing {
    //start et end sont les indices sur la plage du tableau à manipuler .
    //arg = null ou éventuel argument nécessaire à un calcul (ex: arg=moyenne pour calcul de variance)
    double basicCompute(double[] numbers, int start, int end, Double arg); //sum or average or ....
}

```

***ResultAggregate.java***

```

package tp.thread.sam;
//interface d'une référence de fonction pour recombinaison 2 sous sommes ou 2 sous moyennes
public interface ResultAggregate {
    Double composeTotalRes(double res_tab1, int taille_tab1, double res_tab2, int taille_tab2);
}

```

***MyStatsAlgo.java***

```

package tp.thread;

import tp.thread.sam.ResultAggregate;
import tp.thread.sam.SequentialComputing;

//calcul de somme , moyenne ou variance sur grands tableaux
public class MyStatsAlgo {
    public static final int DECOMP_MIN_SIZE=1024*50;

    public static final Double composeMoyenneTotale(double moyenne_tab1, int taille_tab1,
                                                    double moyenne_tab2, int taille_tab2) {
        return (moyenne_tab1*taille_tab1 + moyenne_tab2*taille_tab2) / (taille_tab1 + taille_tab2);
    }

    //versions ordinaires (sans decomposition en 2 sous parties):

    static Double moyenne_ordinaire_subPart(double[] tableau, int deb, int fin, Double notUsedArg){
        Double somme = 0.0;
        for(int i=deb; i<=fin ;i++) {
            somme += tableau[i];
        }
        int sizeSubPart = (fin - deb)+1;
        return somme / sizeSubPart;
    }

    static Double variance_ordinaire_subPart(double[] tableau, int deb, int fin, Double moyenne){

```

```

        Double varianceFoisN = 0.0;
        for(int i=deb; i<=fin ;i++) {
            varianceFoisN += (tableau[i] - moyenne) * (tableau[i] - moyenne);
        }
        int sizeSubPart = (fin - deb)+1;
        return varianceFoisN/sizeSubPart;
    }

//versions avec decomposition en 2 sous parties:

    static double moyenne_decomp_subPart(double[] tableau,int deb,int fin){
        return compute_decomp_subPart(tableau,deb,fin,null,
            MyStatsAlgo::moyenne_ordinaire_subPart,MyStatsAlgo::composeMoyenneTotale);
    }

    static double variance_decomp_subPart(double[] tableau,int deb,int fin,double moyenne){
        return compute_decomp_subPart(tableau,deb,fin,moyenne,
            MyStatsAlgo::variance_ordinaire_subPart,MyStatsAlgo::composeMoyenneTotale);
    }

    static double compute_decomp_subPart(double[] tableau,int deb,int fin,Double arg,
        SequentialComputing seqComputing , ResultAggregate resAggregate){
        double resSp;
        int sizeSubPart = (fin - deb)+1;
        if(sizeSubPart >= DECOMP_MIN_SIZE){
            int indiceMilieu = deb + (sizeSubPart / 2);
            double resCalculSousPartie1 = compute_decomp_subPart(tableau,deb,indiceMilieu-1,arg,seqComputing,resAggregate);
            double resCalculSousPartie2 = compute_decomp_subPart(tableau,indiceMilieu,fin,arg,seqComputing,resAggregate);
            resSp = resAggregate.composeTotalRes(resCalculSousPartie1 , (indiceMilieu - deb) ,
                resCalculSousPartie2 , (fin - indiceMilieu +1) );
        }
        else
            resSp = seqComputing.basicCompute(tableau,deb,fin,arg);
        return resSp;
    }

//fonctions de niveau principal (appels simples , niveau global):
    static double moyenne_ordinaire(double[] tableau){
        return moyenne_ordinaire_subPart(tableau, 0, tableau.length - 1 , null );
    }

    static double ecartType_ordinaire(double[] tableau){
        double moyenne = moyenne_ordinaire_subPart(tableau, 0, tableau.length - 1 , null );
        double variance = variance_ordinaire_subPart(tableau, 0, tableau.length - 1 , moyenne );
        return Math.sqrt(variance);
    }

    static double moyenne_decomp(double[] tableau){
        return moyenne_decomp_subPart(tableau, 0, tableau.length - 1 );
    }

    static double ecartType_decomp(double[] tableau){
        double moyenne = moyenne_decomp_subPart(tableau, 0, tableau.length - 1 );
        double variance = variance_decomp_subPart(tableau, 0, tableau.length - 1 , moyenne);
        return Math.sqrt(variance);
    }
}

```



**MyStatsMultiProc.java**

```

package tp.thread;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

import tp.thread.sam.ResultAggregate;
import tp.thread.sam.SequentialComputing;

public class MyStatsMultiProc extends RecursiveTask<Double> {
    private static final int FORK_JOIN_MIN_SIZE=1024*50;
        // THRESHOLD/SEUIL à éventuellement ajuster selon complexité du calcul
    private double[] tab;
    private int start,end;
    private SequentialComputing seqComputing; //référence de fonction pour calcul ordinaire de somme ou moyenne ou ...
    private ResultAggregate resAggregate; //référence de fonction pour recombinaison 2 sous sommes ou 2 sous moyennes
    private Double arg; //null ou éventuel argument nécessaire à un calcul (ex: arg=moyenne pour calcul de variance)

    public MyStatsMultiProc(double[] tableau,int deb,int fin , Double arg,
        SequentialComputing seqComputing , ResultAggregate resAggregate){
        this.tab = tableau; this.start = deb; this.end = fin; this.arg = arg;
        this.seqComputing = seqComputing; this.resAggregate = resAggregate;
    }

    static double moyenne_multiProc(double[] tableau){
        MyStatsMultiProc myStatsMultiProc= new MyStatsMultiProc(tableau, 0, tableau.length - 1,
            null,MyStatsAlgo::moyenne_ordinaire_subPart,MyStatsAlgo::composeMoyenneTotale);
        ForkJoinPool threadPool = new ForkJoinPool();
        return threadPool.invoke(myStatsMultiProc);
    }

    static double ecartType_multiProc(double[] tableau){
        double moyenne = moyenne_multiProc(tableau);
        MyStatsMultiProc myStatsMultiProc= new MyStatsMultiProc(tableau, 0, tableau.length - 1,
            moyenne,MyStatsAlgo::variance_ordinaire_subPart,MyStatsAlgo::composeMoyenneTotale);
        ForkJoinPool threadPool = new ForkJoinPool();
        double variance = threadPool.invoke(myStatsMultiProc);
        return Math.sqrt(variance);
    }

    @Override
    protected Double compute() {
        Double res =0.0;
        //System.out.println("MyStatsMultiProc.compute() executé par "+Thread.currentThread().getName() );

        //NB: etant donné que la version forkJoin ajoute une complexité au niveau du code
        // (instance à créer , thread à gérer) , cette version ne sera activée/utilisée
        //que pour traiter des sous tableaux dont la taille minimum est supérieure à FORK_JOIN_MIN_SIZE

        int sizeSubPart = (end - start)+1;

        if(sizeSubPart >= FORK_JOIN_MIN_SIZE) {
            int indiceMilieu = this.start + (sizeSubPart / 2);
            // pas de parametre dans .compute() et donc les tab et indices doivent être renseignés
            //en tant qu'attributs + constructeurs
            MyStatsMultiProc sousCalculGaucheViaForkJoin=
                new MyStatsMultiProc(tab,start,indiceMilieu-1,arg,seqComputing,resAggregate);

```

```

MyStatsMultiProc sousCalculDroitViaForkJoin=
    new MyStatsMultiProc(tab,indiceMilieu,end,arg,seqComputing,resAggregate);

sousCalculGaucheViaForkJoin.fork();//déléguer (via potentiel autre thread)

//sous solution A (.compute() ) :
//Double resCalculSousPartie2 = sousCalculDroitViaForkJoin.compute();//faire soit même (via meme thread)

//sous solution B (.fork/join ) :
sousCalculDroitViaForkJoin.fork();//déléguer (via potentiel autre thread)
Double resCalculSousPartie2 = sousCalculDroitViaForkJoin.join(); //attendre

Double resCalculSousPartie1 = sousCalculGaucheViaForkJoin.join(); //attendre

res = resAggregate.composeTotalRes(resCalculSousPartie1 , (indiceMilieu - start) ,
                                   resCalculSousPartie2 , (end - indiceMilieu +1) );
} else {
    res = seqComputing.basicCompute(this.tab,this.start,this.end,this.arg);
}
return res;
}

```

Exemple de performances comparées (sur un petit i7 ) :

nb processors:4

tableau de taille = 134217728

## moyenne ordinaire (sans decomposition)

##MOYENNE=500.0102357180581

## **601 ms**

\$\$ moyenne avec decomposition

\$\$MOYENNE=500.01023571807764

\$\$ **174 ms**

\*\* moyenne avec decomposition optimise pour machine multi-processeurs (fork/join)

\*\*MOYENNE=500.01023571807764

\*\* **64 ms**

## 7. CompletableFuture , streams asynchrones

**CompletableFuture<T>** disponible depuis le *jdk 1.8* est une version améliorée de **Future<T>** pour **enchaîner certains traitements asynchrones** et qui comporte *quelques similitudes avec les "callback" et "Promise.then()" de javascript* .

```
public class CompletableFuture<T>
extends Object
implements Future<T>, CompletionStage<T>
```

dans le package **java.util.concurrent** .

Dans les grandes lignes , la **principale valeur ajoutée de CompletableFuture<T>** vis à vis de **Future<T>** réside dans l'**implémentation de l'interface CompletionStage<T>** de manière à **bien resynchroniser / ordonner différentes tâches asynchrones qui seront par nature exécutées en différé** (dans le futur) .

Autrement dit , un bloc d'instructions de ce type :

```
CompletableFuture.supplyAsync(traitementAsynchrone1 )
    .thenApply( traitementAsynchrone2 )
    .thenApply(traitementAsynchrone3)
    .thenAccept( traitementAsynchroneFinal );
```

correspond à enregistrement de tâches asynchrones à effectuer en différé dès que possible .

Le future résultat du *traitementAsynchrone1* constituera l'entrée de la tâche *traitementAsynchrone2* qui ne pourra alors être démarrée que lorsque la tâche *traitementAsynchrone1* sera terminée et ainsi de suite via cet enchaînement de `.thenApply()` `.then...()`

### Rappel :

- l'interface (uni-)fonctionnelle **Supplier<T>** comporte l'unique méthode **T .get()** et correspond à un producteur ou fournisseur de valeur de Type *T* .
- l'interface (uni-)fonctionnelle **Function<T,R>** comporte la méthode **R .apply(T t)** et correspond à une fonction appliquée à un unique paramètre de type *T* et retournant *R* .
- l'interface (uni-)fonctionnelle **Consumer<T>** comporte la méthode **void .accept(T t)** et correspond à un consommateur d'élément de type *T* (sans valeur produite) .

**Principales méthodes de CompletableFuture<T> et CompletionStage<T> :**

NB1 : l'abréviation **CF<T>** est à comprendre comme **CompletableFuture<T>**

**NB2 :** Sans précision de l'executor (souvent en tant que dernier paramètre facultatif des méthodes surchargées) , la classe CompletableFuture lance en interne les tâches asynchrones via **ForkJoinPool.commonPool()** par défaut .

**NB3 :** Au sein de ce tableau , les méthodes en *italiques* seront à considérées comme "**static**" .  
et <T,R> est généralement à interpréter comme < ? super T , ? extends R>

CF<Void> <b>runAsync</b> (Runnable r)	retourne un CF<T> correspondant à la future exécution asynchrone de la tâche r (Runnable)
CF<U> <b>supplyAsync</b> ( Supplier<U> s)	retourne un CF<U> correspondant à la future exécution asynchrone de la tâche s (fournissant une valeur de type U)
CF<R> <b>thenApply</b> (Function<T,R> f)	Souvent en milieu d'enchaînement, après la fin du CF<T> précédent (vu comme préfixe (this)) , retourne un CF<R> correspondant à la future exécution asynchrone de la tâche fonctionnelle f
CF<R> <b>thenCompose</b> ( Function<T,CF<R>> f)	Un peu comme thenApply() mais en précisant une fonction qui (immédiatement, sans attente) construit elle même une instance de CF <R> qui sera consommée dans la suite de l'enchaînement
CF<T> <b>exceptionally</b> ( Function<Throwable ex,T> f)	Souvent en milieu d'enchaînement, transforme une exception potentielle en élément de type T pour le bon déroulement de la suite des enchaînements, retransmet la valeur inchangée si pas d'exception .
CF<R> <b>handle</b> (BiFunction<T, Throwable,R> f)	Combine le comportement de thenApply() et exceptionally() via une tâche asynchrone prenant 2 argument en entrée : value et exception .
CF<Void> <b>thenAccept</b> (Consumer<T> c)	En terminaison, après la fin du CF<T> précédent (vu comme préfixe (this)) , retourne un CF<Void> correspondant à la future exécution asynchrone de la tâche terminale c (Consumer avec argument en entrée)
CF<Void> <b>thenRun</b> (Runnable r)	Souvent en terminaison, après la fin du CF<T> précédent (vu comme préfixe (this)) , retourne un CF<Void> correspondant à la future exécution asynchrone de la tâche terminale r (Runnable sans argument en entrée)
boolean <b>complete</b> (T value) et boolean <b>completeExceptionally</b> ( Throwable ex)	si tâche pas terminée , fixe le résultat de cette tâche qui sera ultérieurement récupérée par .get().

**Remarque :** si cf.**thenApplyAsync**( ... ); à la place de cf.**thenApply**( ... ); alors tâche quelquefois exécutée par encore un autre thread.

## 7.1. Premiers exemples de CompletableFuture

### EssaiAsyncJava8.java

```
package tp.langage.thread;

import java.util.concurrent.CompletableFuture; import java.util.function.Supplier;

public class EssaiAsyncJava8 {
    public static void main(String[] args) {
        System.out.println("debut main / interpreted by " + Thread.currentThread().getName());

        //initialisation asynchrone:
        //CompletableFuture<Void> cf = CompletableFuture.runAsync(aRunnableObject); //with no return result !!!
        CompletableFuture<Double> completableFuture1 =
            /*CompletableFuture.supplyAsync( new Supplier<Double>(){
                public Double get(){ LongTask.simulateLongTask("long computing - p1", 2000);
                return 2.0; }
            });*/
        CompletableFuture.supplyAsync( ()-> { LongTask.simulateLongTask("long ...- p1", 2000);
            /*throw new RuntimeException("exceptionXY");*/ return 2.0; } );

        //En cas d'exception en asynchrone/tâche de fond:
        CompletableFuture<Double> safecompletableFuture1 =
            completableFuture1.exceptionally(ex -> { System.out.println("problem: " +
                ex.getMessage()); return 0.0; } );
        /*CompletableFuture<Double> safecompletableFuture1 =
            completableFuture1.handle((resOk,ex) -> {
                if(resOk!=null) return resOk;
                else { System.out.println("problem: " + ex.getMessage()); return 0.0; } } );*/

        System.out.println("suite main A / interpreted by " + Thread.currentThread().getName());

        // continuations asynchrones (avec Function<T1,T2>):
        CompletableFuture<Double> completableFuture2 =
            safecompletableFuture1.thenApply((x) -> { LongTask.simulateLongTask("long computing -
                p2", 2000); return x*x; } );

        CompletableFuture<String> completableFuture3 =
            completableFuture2.thenApply((x) -> { LongTask.simulateLongTask("long computing - p3",
                2000); return String.valueOf(x); } );

        System.out.println("suite main B / interpreted by " + Thread.currentThread().getName());
        //fin/terminaison asynchrone:
        completableFuture3.thenAccept((x) -> System.out.println(x) );
    }
}
```

```
//completableFuture.thenRun(()->System.out.println("ok")); //with no input !!!

LongTask.simulateLongTask("pause pour eviter arrêt complet du programme " +
    "avant la fin des taches de fond" , 8000);
System.out.println("fin main / interpreted by " + Thread.currentThread().getName());
}
}
```

**Résultats :**

```
debut main / interpreted by main
suite main A / interpreted by main
>>(begin)long computing - p1 / by ForkJoinPool.commonPool-worker-3
suite main B / interpreted by main
>>(begin)pause pour eviter arrêt complet du programme avant la fin des taches de fond / by main
<<(end)long computing - p1 / by ForkJoinPool.commonPool-worker-3
>>(begin)long computing - p2 / by ForkJoinPool.commonPool-worker-3
<<(end)long computing - p2 / by ForkJoinPool.commonPool-worker-3
>>(begin)long computing - p3 / by ForkJoinPool.commonPool-worker-3
<<(end)long computing - p3 / by ForkJoinPool.commonPool-worker-3
4.0
<<(end)pause pour eviter arrêt complet du programme avant la fin des taches de fond / by main
fin main / interpreted by main
```

Variation syntaxique (générant le même résultat) :**EssaiAsyncJava8V2.java**

```
package tp.langage.thread;

import java.util.concurrent.CompletableFuture;

public class EssaiAsyncJava8V2 {

    public static Double extractInitValue() {
        LongTask.simulateLongTask("long computing - p1" , 2000);
        /*throw new RuntimeException("exceptionXY");*/ return 2.0;
    }

    public static Double auCarre(Double x){
        LongTask.simulateLongTask("long computing - p2" , 2000); return x*x;
    }

    public static String convertAsString(Double x){
        LongTask.simulateLongTask("long computing - p3" , 2000); return String.valueOf(x);
    }
}
```

```

public static void displayString(String s){
    System.out.println(s) ;
}

public static void main(String[] args) {
    System.out.println("debut main / interpreted by " + Thread.currentThread().getName());

    CompletableFuture.supplyAsync(EssaiAsyncJava8V2::extractInitValue )
        .exceptionally(ex -> { System.out.println("problem: " +
            ex.getMessage()); return 0.0; } )
        .thenApply(EssaiAsyncJava8V2::auCarre )
        .thenApply(EssaiAsyncJava8V2::convertAsString)
        .thenAccept(EssaiAsyncJava8V2::displayString );

    System.out.println("suite main / interpreted by " + Thread.currentThread().getName());
    LongTask.simulateLongTask("pause pour eviter arrêt complet du programme" +
        " avant la fin des taches de fond" , 8000);
    System.out.println("fin main / interpreted by " + Thread.currentThread().getName());
}
}

```

#### Autre variation syntaxique avec this et lambda expressions :

##### EssaiAsyncJava8V2Bis.java

```

package tp.langage.thread;
import java.util.concurrent.CompletableFuture;
public class EssaiAsyncJava8V2Bis {
    public static void simuLong(String number,long nbMs) {
        LongTask.simulateLongTask("long computing - p"+number , nbMs);
    }
    private double initialXValue;
    private String result;

    public void noStaticMethodWithLambda() {
        this.initialXValue=2;
        //code am"lirable avec synchronized(this){ this... }
        CompletableFuture.supplyAsync(()->{ simuLong("p1",2000);
            return this.initialXValue; } )
            .thenApply( (x)->{ simuLong("p2",2000); return x*x; } )
            .thenApply( (x)->{ simuLong("p3",2000); return String.valueOf(x); } )
            .thenAccept( (s)->{System.out.println(s); this.result=s});
        LongTask.simulateLongTask("pause pour eviter arrêt complet du programme"
            + " avant la fin des taches de fond" , 8000);
    }
}

```

```

        System.out.println("result="+this.result);
    }

    public static void main(String[] args) {
        EssaiAsyncJava8V2Bis thisApp = new EssaiAsyncJava8V2Bis();
        thisApp.noStaticMethodWithLambda();
    }
}

```

*result=4*

## 7.2. Combinaisons avec CompletableFuture

Exemple partiel : *EssaiAsyncJava8V3.java*

```

package tp.langage.thread;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

public class EssaiAsyncJava8V3 {

    //plein de code commun avec EssaiAsyncJava8V2 (pas répété ici)

    public static Double plus(Double x, Double y){
        LongTask.simulateLongTask("long computing - plus" , 2000); return x+y;
    }

    public static CompletionStage<Double> cflnitVal() {
        return CompletableFuture.supplyAsync(EssaiAsyncJava8V3::extractInitValue );
    }

    //méthode static pour pour then.compose(...) :
    public static CompletionStage<String> cfAsString(Double x){
        CompletableFuture<Double> cfDouble = new CompletableFuture<Double>();
        cfDouble.complete(x);
        System.out.println("**** tout le debut de cfAsString sans attente ****");
        return cfDouble.thenApplyAsync(EssaiAsyncJava8V3::convertAsString);
    }

    public static void main(String[] args) {
        System.out.println("debut main de EssaiAsyncJava8V3");
        //thenCompose register another future to apply/compose to thisFuture (without waiting)
        // to produce a new future :
        cflnitVal().thenCompose(EssaiAsyncJava8V3::cfAsString)
            .thenAccept(EssaiAsyncJava8V3::displayString );

        CompletableFuture<Double> cfD1 = new CompletableFuture<Double>();
    }
}

```



```

cfD1.complete(3.0); //COMPLETE BY MAIN --> NEED .thenApplyAsync() to be continued by other thread

CompletableFuture<Double> cfD2 = new CompletableFuture<Double>();
cfD2.complete(2.0); //COMPLETE BY MAIN --> NEED .thenApplyAsync() to be continued by other thread
System.out.println("suite du main de EssaiAsyncJava8V3");
CompletableFuture<Double> cfD3 = cfD1.thenApplyAsync(EssaiAsyncJava8V3::auCarre);//3*3=9
CompletableFuture<Double> cfD4 = cfD2.thenApplyAsync(EssaiAsyncJava8V3::auCarre);//2*2=4

//thenCombine apply a biFunction from 2 futures (this & other) to produce a new future

CompletableFuture<Double> cfD5 =
    cfD3.thenCombine(cfD4, EssaiAsyncJava8V3::plus);//9+4=13
cfD5.thenAccept((x)->System.out.println(x));

CompletableFuture<Double> cfD6 = cfD1.thenApplyAsync(EssaiAsyncJava8V3::auCarre);//3*3=9
CompletableFuture<Double> cfD7 = cfD2.thenApplyAsync(EssaiAsyncJava8V3::auCarre);//2*2=4
cfD6.thenAcceptBoth(cfD7, (x,y) -> System.out.println(x*y));//9*4 = 36

LongTask.simulateLongTask("pause pour éviter arrêt complet du programme "
    + "avant la fin des taches de fond", 8000);
System.out.println("fin main de EssaiAsyncJava8V3 / interpreted by " + Thread.currentThread().getName());
}

```

### Autres combinaisons :

#### Variantes proches de thenCombine :

**.thenAcceptBoth(otherFuture, biConsumer )** --> action terminale (sans valeur calculée ni retournée) avec valeurs en entrées lorsque les 2 futures sont terminés

**.runAfterBoth(otherFuture, runnable )** --> action sans valeur en entrée lorsque les 2 futures sont terminés

#### Variantes proches de thenAcceptBoth/runAfterBoth :

**.thenAcceptEither(otherFuture, consumer )** --> action avec valeur en entrées lorsque le premier des 2 futures est terminé

**.runAfterEither(otherFuture, runnable )** --> action sans valeur en entrée lorsque le premier des 2 futures est terminé

#### Variante proche de thenAcceptEither :

**thenApplyEither(otherFuture , function)** --> génère un nouveau future (avec valeur) pour poursuivre un enchaînement avec thenApply() ou autre.

#### Variantes avec nombres d'arguments variables:

**.allOf(...)** pour attendre la fin de n(3 ou plus) futures

**.anyOf(...)** pour attendre la fin du plus rapide parmi n

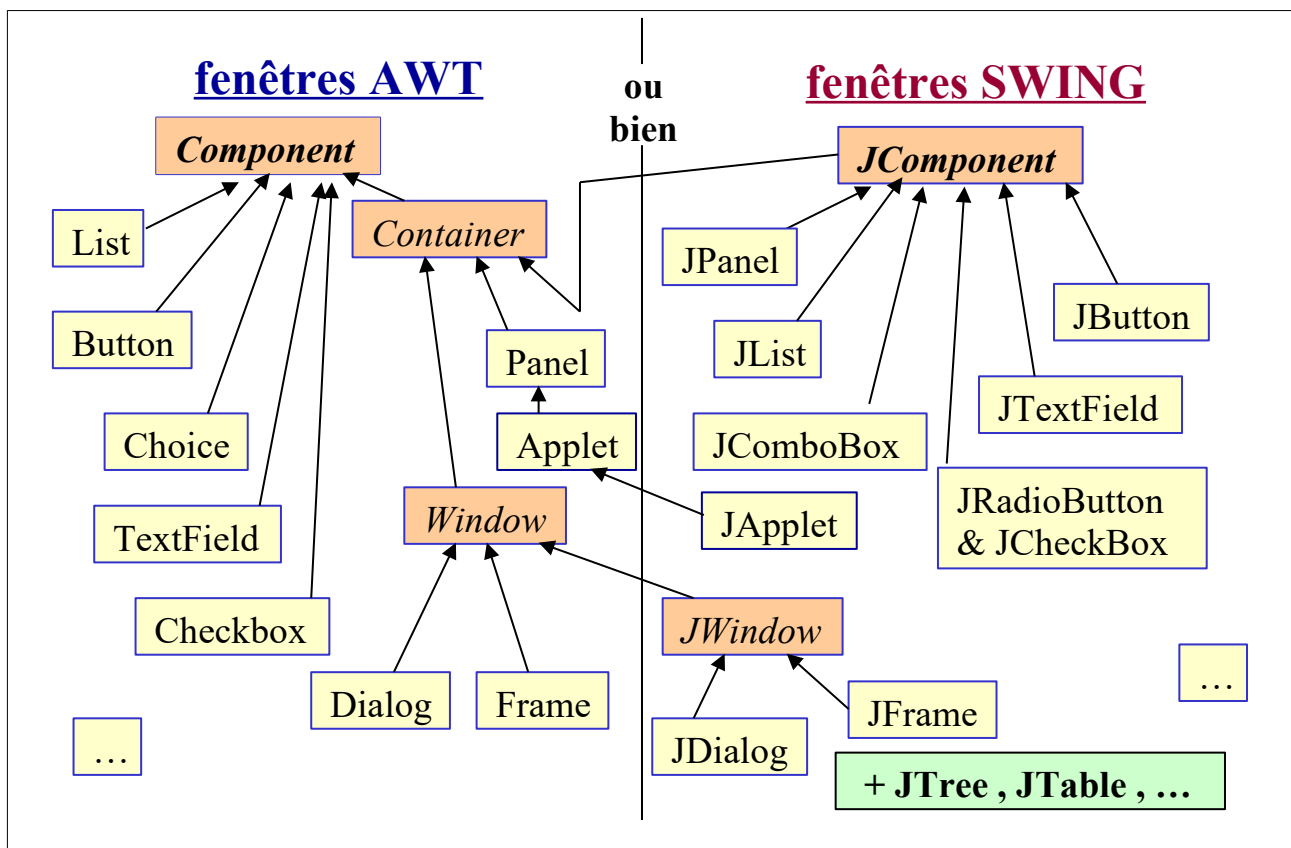
# XI - Api pour IHM/GUI (swing, ...)

## 1. Éléments 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:

<b>getParent()</b>	retourne le Container contenant l'objet
<b>setBackground(color)</b>	Fixe la couleur de fond
<b>setForeground(color)</b>	Fixe la couleur d'avant plan (texte,traits,...)
<b>setFont(fonte)</b>	Fixe la police de caractères
<b>setVisible(booléen)</b>	cache ou rend visible l'objet graphique
<b>setEnabled(booléen)</b>	active ou inactive (grise) l'objet
<b>setLocation(x,y), setSize(w,h)</b>	déplace , redimensionne l'objet
<b>requestFocus()</b>	demande le focus sur cet objet

<b>bounds()</b>	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'éditations , ...) 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:

<b>add(component)</b>	Ajoute un composant (contrôle,panel, ...)
<b>remove(component)</b>	Enlève un sous objet
<b>countComponents()</b>	Retourne le nombre de composants
<b>setLayout(layoutMgr)</b>	Associe un gestionnaire de répartition au conteneur
<b>locate(x,y)</b>	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:

<b>setTitle(titre)</b>	Change le titre
<b>setMenuBar(barreDeMenu)</b>	Associe un menu à la fenêtre
<b>setResizable(monBooleen)</b>	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:

<i>classe</i>	<i>fonctionnalités</i>
<b>JTextArea</b>	zone de texte simple à plusieurs lignes (light).
<b>TextPane</b>	éditeur de texte sophistiqué (mise en forme possible des caractères).
<b>EditorPane</b>	Zone de texte sophistiquée (text/plain ou text/html )

## Boîte de dialogue

- `javax.swing.JDialog` est la classe générique des boîtes 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:

<code>.setToolTipText( "Chaine InfoBulle" )</code>	précise le texte de la bulle d'aide
<code>.setVisible(true/false)</code>	montre ou cache le composant
<code>.setEnabled(false/true)</code>	grise ou dégrise le composant
<code>.setOpaque(false)</code>	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. <b>showMessageDialog</b> (p,"Bienvenue");	Affiche une boîte de message
name= JOptionPane. <b>showInputDialog</b> (p,"Quel est votre nom");	Prompt
JOptionPane. <b>showConfirmDialog</b> (p,"voulez vous ...?") => renvoie une valeur du genre OK_OPTION ou YES_OPTION	Demande de confirmation (Yes,No,Cancel)

Paramètres optionnels 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);

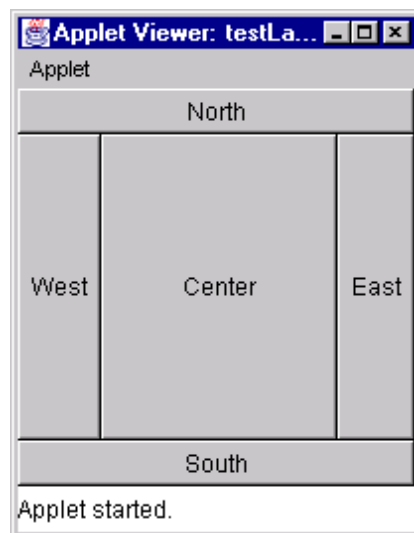
Celui-ci sera alors automatiquement utilisé par la suite.

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:

<b>BorderLayout</b>	<i>Arrange les contrôles sur les bords et au centre du</i>
---------------------	--

	<p><b>conteneur</b> en fonction des indications "South", "North", "East", "West", "Center" .</p> <p>Les paramètres facultatifs <i>hgap</i> et <i>vgap</i> du constructeur permettent de préciser l'espacement entre deux composants adjacents.</p>
<p><b>FlowLayout</b> (par défaut)</p>	<p>Arrange les composants en ligne de gauche à droite.</p> <p>Fait entrer autant de composants que possible sur la ligne courante avant de passer à la prochaine.</p> <p>Au sein d'une ligne les choses sont alignées en fonction du paramètre facultatif <i>align</i> du constructeur.</p> <p>FlowLayout.CENTER , .LEFT , .RIGHT</p>
<p><b>GridLayout,</b> <b>CardLayout,</b> <b>GridBagLayout,</b> ...</p>	<p>... autres gestionnaires (complexes)...</p>



### 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 choses 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:

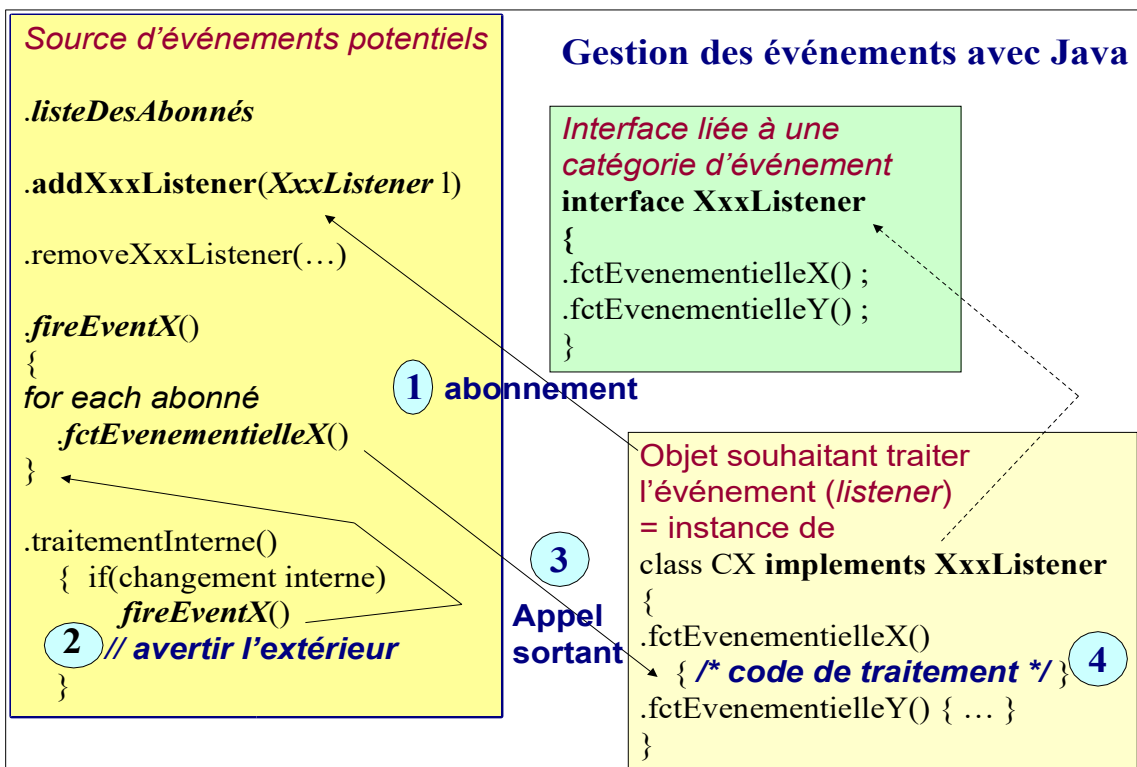
```
composantSource.addXXXListener(objetQuiSouhaiteGererEvenement);
```

```
composantSource.addXXXListener(deuxièmeAbonné);
```

>>>> *XXX correspond ici 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 Delegate 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()
    {
        Delegate unDelegate = new Delegate();
        JButton monBouton = new JButton("Touch Me");
        monBouton.addActionListener(unDelegate); // abonnement
        this.getContentPane().add(monBouton);
    }
    public static void main(String [] argv)
    {
        JFrame f = new PetiteFenetre();
        f.pack();
        f.setVisible(true);
    }
}
```

```
class Delegate implements ActionListener
{
    ...    /* il faut coder toutes les méthodes de l'interface ActionListener
           → soit une seule méthode ici (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) :

<i>Interface / catégorie d'événements</i>	<i>types de composants sources</i>
<b>ActionListener</b>	Button (éventuellement JRadioButton, JCheckBox, ...)
<b>MouseListener</b>	Panel, Canvas , ...(éventuellement JList, ...)
<b>KeyListener</b>	Zone de saisie (==> codez en priorité <i>keyReleased</i> )
<b>ItemListener</b>	Boutons radios , cases à cocher , zones de listes, ...
...	...

<i>Interface</i>	<i>méthodes à implémenter (void)</i>
<b>ActionListener</b>	<b>actionPerformed</b> (ActionEvent)
<b>AdjustmentListener</b>	ajustmentValueChanged(AjustmentEvent)
<b>ComponentListener</b>	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) <b>componentResized</b> (ComponentEvent)
<b>ContainerListener</b>	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
<b>FocusListener</b>	focusGained(FocusEvent) focusLost(FocusEvent)
<b>ItemListener</b>	<b>itemStateChanged</b> (ItemEvent)
<b>KeyListener</b>	keyPressed(KeyEvent) <b>keyReleased</b> (KeyEvent) keyTyped(KeyEvent)

<b>MouseListener</b>	<b>mouseClicked</b> (MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) <b>mousePressed</b> (MouseEvent) <b>mouseReleased</b> (MouseEvent)
<b>MouseMotionListener</b>	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
<b>TextListener</b>	textValueChanged(TextEvent)
<b>WindowListener</b>	windowActivated(WindowEvent) <b>windowClosed</b> (WindowEvent) windowClosing(WindowEvent) windowDesactivated(WindowEvent) windowDeiconified(WindowEvent) 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

## XII - Annexe – nouveautés v12-17

### 1. Switch expressions

La nouvelle forme du **switch()** avec des "**lambda**" était apparue dès les jdk 1.12 et 1.13 en mode "preview". C'est enfin devenu une nouvelle fonctionnalité "standard" à partir de la version **1.14**

#### 1.1. expression d'un switch/case avec des "lambda"

```
int dayOfWeek = 4;
System.out.print("dayOfWeek="+dayOfWeek + " -> ");
switch(dayOfWeek) {
    case 1 -> System.out.println("monday");
    case 2 -> System.out.println("tuesday");
    case 3 -> System.out.println("wednesday");
    case 4 -> System.out.println("thursday");
    case 5 -> System.out.println("friday");
    case 6 -> System.out.println("saturday");
    case 0 , 7 -> System.out.println("sunday");
    default -> System.out.println("unknown");
}
```

Résultat:

```
dayOfWeek=4 -> thursday
```

#### 1.2. switch en tant qu'expression retournant un résultat

```
String dayName="vendredi";
System.out.print("dayName="+dayName + " -> ");
int dayNumber =
    switch(dayName) {
        case "lundi" , "monday" -> 1;
        case "mardi" , "tuesday" -> 2;
        case "mercredi" , "wednesday" -> 3;
        case "jeudi" , "thursday" -> 4;
        case "vendredi" , "friday" -> 5;
        case "samedi" , "saturday" -> 6;
        case "dimanche" , "sunday" -> 7;
        default -> 0;
    };
System.out.println("dayNumber="+dayNumber);
```

Résultat :

```
dayName=vendredi -> dayNumber=5
```

### 1.3. switch avec expressions complexes et mot clef yield

Lorsqu'une expression doit nécessiter plusieurs instructions pour être calculée, on peut englober celles-ci par un bloc d'accolades et préciser la valeur calculée/retournée via le mot clef "yield" (ressemblant à un "return" mais de niveau switch et pas de niveau fonction) .

```
int value = (int) (Math.random() * 14);
String result = switch( value ) {
    case 0, 2, 4, 6, 8 -> {
        double racine = Math.sqrt( value );
        yield "chiffre pair dont la racine carré vaut " + racine;
    }
    case 1, 3, 5, 7, 9 -> {
        double carre = value * value;
        yield "chiffre impair dont le carré vaut " + carre;
    }
    default -> "ce n'est plus un chiffre, mais un nombre";
};
System.out.println("value=" + value + "->" + result );
```

Résultats :

value=2->chiffre pair dont la racine carré vaut 1.4142135623730951

value=13->ce n'est plus un chiffre, mais un nombre

value=3->chiffre impair dont le carré vaut 9.0

...



## 2. Pattern Matching instanceof

En preview en version 14, 15 et standardisé en version 16 et 17/LTS .

```
public class TestPatternMatchingInstanceOfApp {

    public static void main(String[] args) {
        String s = "abc";
        oldStyleWithoutPatternMatchingInstanceOf(s);
        newStyleWithPatternMatchingInstanceOf(s);
    }

    public static void oldStyleWithoutPatternMatchingInstanceOf(Object obj) {
        if (obj instanceof String) {
        String str = (String) obj;
        int len = str.length();
        System.out.println("obj is as string of length="+len);
        }
    }

    public static void newStyleWithPatternMatchingInstanceOf(Object obj) {
        if (obj instanceof String str) {
        int len = str.length();
        System.out.println("obj is as string of length="+len);
        }
    }
}
```

C'est un **sucre syntaxique** (simplification de syntaxe) **bien pratique** .

### 3. TextBloc

Un "TextBloc" (depuis java 15) est une chaîne de caractère formulée comme un **bloc de texte sur plusieurs lignes**. Encadré par des `"""` (triple guillemet), un *textBloc* peut comporter des caractères quelconques qui n'ont plus besoin d'échappement.

Ceci est surtout utile pour le format JSON (nécessitant des `"` et pas de `'`) à l'intérieur d'une chaîne java délimitée par des `"`.

*écriture sans textbloc et très peu lisible :*

```
public static void oldStyleWithoutTextBloc() {
    String myJsonString
        = "{\r\n" + "\"username\" : \"JeanBon\", \r\n" + "\"country\" : \"France\" \r\n" + "}";
    System.out.println("myJsonString="+myJsonString);
}
```

*équivalent bien plus lisible avec textBloc :*

```
public static void newStyleWithTextBloc() {
    String myJsonStringAsTextBloc = """
        {
            "username" : "AlexTherieur",
            "country" : "Belgique"
        }
        """;
    System.out.println("myJsonStringAsTextBloc="+myJsonStringAsTextBloc);
}
```

→ affiche

```
myJsonStringAsTextBloc={
"username" : "AlexTherieur",
"country" : "Belgique"
}
```

```
public static void someTextBlocs() {
    //au minimum 2 lignes de code :
    //  """ suivi de newligne pour commencer
    //  texte quelconque au milieu
    //  """; pour finir
    String simpleTextBloc0 = """
        blabla""";
    System.out.println("simpleTextBloc0="+simpleTextBloc0);
    //affiche simpleTextBloc0=blabla
}
```

```

//pour bien controller l'intentation , il faut savoir que lorsque le compilateur va analyser
//le code source, il va supprimer le nombre minimum de tabulations qu'il y a
//au niveau de toutes les lignes de code (du textBloc sans tenir compte du tout début = "")
//Bonne pratique : au sens IDE (eclipse/IntelliJ/...) placer des tabulations pour indenter le code
//          au sens contenu final du textBloc , utiliser plutôt des espaces et sauts de lignes
//          placer la fin "" sur une ligne séparée
//          (comportant le nombre de tabulations qui seront explicitement supprimées du texte)
String textBloc3AvecIndentationControlee = ""
    {
        "username" : "AlexTherieur",
        "country" : "Belgique"
    }
    """;

System.out.println("textBloc3AvecIndentationControlee="+textBloc3AvecIndentationControlee);

//par défaut les espaces en fin de ligne sont considérés inutiles et ne sont pas conservés
//le caractère d'échappement \s permet d'explicitement demander à conserver les espaces en fin de ligne
String textBloc4AvecEspacesConserveEnFinDeLignes = ""
    blabla  \s""";

System.out.println("textBloc4AvecEspacesConserveEnFinDeLignes="
    +textBloc4AvecEspacesConserveEnFinDeLignes + "suiteApres");

}

```

```

public static void textBlocsWithFormattedParams() {
    String username="toto";
    int age=40;
    double height = 1.73;
    boolean ok=true;
    String jsonString = ""
        {
            "username" : "%s",
            "age" : %d,
            "height" : %s,
            "ok" : %b
        }
        """.formatted(username,age,String.valueOf(height),ok);
        //same syntax of String.format
    System.out.println("jsonString="+jsonString);
}

```

affiche

```

{ "username" : "toto",
  "age" : 40,
  "height" : 1.73,
  "ok" : true
}

```

## 4. Record (classe de données simplifiée pour DTO)

"Preview" en version 14 et 15, les "record" sont standardisés depuis java 16 (et 17 LTS) .

//exemple de classe POJO/JavaBean classique (pour comparaison) :

```
public class OldCustomer{

    private Integer id;
    private String firstName;
    private String lastName;

    public OldCustomer() {        super();
    }

    public OldCustomer(Integer id, String firstName, String lastName) {
        super();
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return "OldCustomer [id=" + id + ", firstName=" + firstName
            + ", lastName=" + lastName + "]";
    }

    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }

    //+ autres get/set
    //+ hashCode()
    //+ equals()
}
```

classe équivalente avec annotations de lombok :

```
import lombok.*;

@Getter @Setter @ToString
@NoArgsConstructor @AllArgsConstructor
@EqualsAndHashCode
public class LombokCustomer {
    private Integer id;
    private String firstName;
    private String lastName;
}
```

Utilisation classique d'un javaBean codé avec ou sans lombok:

```
OldCustomer oldC1 = new OldCustomer(1,"jean","Bon");
System.out.println("oldC1 as POJO =" + oldC1.toString());
System.out.println("id of oldC1 =" + oldC1.getId());
oldC1.setFirstName("luc");
System.out.println("firstName of oldC1 =" + oldC1.getFirstName());
```

```
OldCustomer oldC2 = new OldCustomer();
System.out.println("oldC2 =" + oldC2.toString());
```

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

```
LombokCustomer lC1 = new LombokCustomer(1,"jean","Bon");
System.out.println("olC1 as Lombok POJO =" + lC1.toString());
System.out.println("id of lC1 =" + lC1.getId());
lC1.setFirstName("luc");
System.out.println("firstName of lC1 =" + lC1.getFirstName());
```

```
LombokCustomer lC2 = new LombokCustomer();
System.out.println("lC2 =" + lC2.toString());
```

Nouveau mot clef "record"

Le nouveau mot clef **record** permet de demander au compilateur de construire automatiquement une sorte de pseudo **POJO IMMUTABLE** (sans Setter possible !) et avec des accesseurs de type .xxx() ne respectant même pas les conventions .getXxx() !!!

Dans les détails : classe construite "final" héritant de **java.lang.Record** et avec des champs privés "final" et avec .equals()/hashCode()/toString() et des accesseurs de type .xxx() et allArgsConstructor .

NB :

- record ne peut pas être vu comme un remplacement de lombok car IMMUTABLE/final/read-only et car .xxx() plutôt que .getXxx()
- record doit plutôt être vu comme une sorte de petit DTO/VO très léger

Minimalist v1 (with no default constructor):

```
public record CustomerRecord(Integer id,String firstName,String lastName) {
}
```

Utilisation :

```
CustomerRecord c1 = new CustomerRecord(1,"jean","Bon");
System.out.println("c1 as record =" + c1.toString());
System.out.println("id of c1 =" + c1.id()); //mais pas .getId() ni .id
c1.setFirstName("luc");
System.out.println("firstName of c1 =" + c1.firstName()); //mais pas .getFirstName() ni .firstName
```

public void **setFirstName**(String firstName) { this.firstName = firstName; }  
 est **interdit/impossible** car **this.firstName** est **final/immutable** !!!

*// V2 (with explicit default constructor) :*

```
public record CustomerRecord(Integer id,String firstName,String lastName) {
    public CustomerRecord() { this(0,null,null);}
    //possible mais pas souvent utile si final/immutable !!!
}
```

Utilisation :

```
CustomerRecord c2 = new CustomerRecord();
System.out.println("c2="+c2.toString());
//affiche ici CustomerRecord[id=0, firstName=null, lastName=null]
```

*V3 (V2 + quelques redefinitions & ajout de methodes )*

```
public record CustomerRecord(Integer id,String firstName,String lastName) {
    public CustomerRecord() { this(0,null,null);}
    public String fullName() { return firstName + " " + lastName;}
    public String lastName() { return lastName.toUpperCase(); }
}
```

*V4 (V3 + new record constructor checking syntax)*

```
public record CustomerRecord(Integer id,String firstName,String lastName) {
    //not useful default constructor:
    public CustomerRecord() { this(0,"unknown","unknown");}

    //new method:
    public String fullName() { return firstName + " " + lastName;}

    //old pojo style getter as a new complementary method :
    public String getFirstName() { return firstName; }

    //accessor redefinition :
    public String lastName() { return lastName.toUpperCase(); }

    //specific record new syntax for allArgsConstructor
    //partial redefinition with value(s) checking:
    public CustomerRecord {
        if (lastName == null || lastName.isBlank()) {
            throw new IllegalArgumentException("lastName is required");
        }
    }
}
```

```

public class TestRecordApp {

    public static void main(String[] args) {
        testUsefulRecordV1();
        testUsefulRecordV2();
        testUsefulRecordV3();
    }

    //private or public locally RECORD seems to be a good use case for "record" new concept :

    private record AddressV1(Integer number,String street,String zipCode,String town) {
        public String toJsonString() {
            return ""
                {
                    "number" : %d,
                    "street" : "%s",
                    "zipCode" : "%s",
                    "town" : "%s"
                }
                """.formatted(number,street,zipCode,town);
        }
    };

    public static void testUsefulRecordV1() {
        AddressV1 a1 = new AddressV1(12,"rueElle","75000" ,"Paris");
        System.out.println("a1="+a1.toString());
        System.out.println("a1="+a1.toJsonString());
    }

    //V2 avec compatibilité avec api jackson-databind (souvent utilisé par JEE , Spring, ...)
    //pratique pour définition de DTO/VO (à la volée)
    public record AddressV2(Integer number,String street,String zipCode,String town) {
    };

    public static void testUsefulRecordV2() {
        AddressV2 a1 = new AddressV2(12,"rueElle","75000" ,"Paris");
        System.out.println("a1="+a1.toString());
        try {
            ObjectMapper jacksonObjectMapper = new ObjectMapper();
            String a1JsonString = jacksonObjectMapper.writeValueAsString(a1);
            System.out.println("via jackson, a1JsonString="+a1JsonString);
        } catch (JsonProcessingException e) {
            e.printStackTrace();
        }
    }
}

```

Résultats :

```

a1=AddressV1[number=12, street=rueElle, zipCode=75000, town=Paris]
a1={

```

```
"number" : 12,
"street" : "rueElle",
"zipCode" : "75000",
"town" : "Paris"
}
```

```
a1=AddressV2[number=12, street=rueElle, zipCode=75000, town=Paris]
via jackson, a1JsonString={"number":12,"street":"rueElle","zipCode":"75000","town":"Paris"}
```

*//NB : Les records de java ne sont bien gérés qu'avec des versions récentes de jackson-databind .  
 // la version 2.12.5 de **jackson-databind** supporte bien les records de java 17 et est compatible  
 //avec une version récente de SpringBoot et SpringMVC (sur partie @RestController)*

```
public class Dto {
    public record Address(Integer number,String street,String zipCode,String town) {
    };

    public record Person(Integer id,String firstName,String lastName,Address address) {
        public Person(Integer id,String firstName,String lastName) {
            this(id,firstName,lastName,null);
        }
    };
}
```

```
public static void testUsefulRecordV3() {
    Dto.Person p1Dto = new Dto.Person(1,"jean","Bon" ,
                                     new Dto.Address(12,"rueElle","75000" ,"Paris"));
    System.out.println("\np1Dto="+p1Dto.toString());

    try {
        ObjectMapper jacksonObjectMapper = new ObjectMapper();
        String p1JsonString = jacksonObjectMapper.writeValueAsString(p1Dto);
        System.out.println("via jackson, p1JsonString="+p1JsonString);
        //-----
        Dto.Person p1BisDto=
            jacksonObjectMapper.readValue(p1JsonString,Dto.Person.class);
        System.out.println("via jackson, p1BisDto=clone de p1Dto="+p1BisDto.toString());
    } catch (JsonProcessingException e) {
        e.printStackTrace();
    }
}
```

### Résultats :

```
p1Dto=Person[id=1, firstName=jean, lastName=Bon, address=Address[number=12,
street=rueElle, zipCode=75000, town=Paris]]
```

```
via jackson, p1JsonString={"id":1,"firstName":"jean","lastName":"Bon","address":
{"number":12,"street":"rueElle","zipCode":"75000","town":"Paris"}}
```

```
via jackson, p1BisDto=clone de p1Dto=Person[id=1, firstName=jean, lastName=Bon,
address=Address[number=12, street=rueElle, zipCode=75000, town=Paris]]
```



## 5. Sealed classes

"Preview" en versions 15,16 et standardisé en version 17 .

```
//NB: "SEALED" signifiant scellé en francais
//est une nouveauté du langage java permettant d'indiquer que seules certaines classes
//(précisées par permits ... , ... )
//auront le droit d'hériter de la classe ou interface actuelle
//cela ressemble à final class qui interdit carrément tout héritage

//sealed keyword can be used if option "enabled preview features for java 16"
public sealed interface AnimalDomestique permits Chat, Chien {
    public void sayHello();
    //...
}
// effet : seules les classes Chat et Chien auront le droit d'hériter de AnimalDomestique .

/*
//HERITAGE via extends ou implements INTERDIT via sealed sur AnimalDomestique :
final class Ce implements AnimalDomestique{
//...
}
*/
```

```
//déclarée final , la classe Chat ne peut pas avoir de sous classe
public final class Chat implements AnimalDomestique{

    @Override
    public void sayHello() {
        System.out.print("chat miaou - ");
    }

    public void ronronner() {
        System.out.println(" ronron ");
    }

}
```

```
//déclarée non-sealed , la classe Chien peut pas avoir des sous classes
public non-sealed class Chien implements AnimalDomestique{
    @Override
    public void sayHello() {
        System.out.print("chien wouf wouf - ");
    }

    public void aLaNiche() {
        System.out.println(" dodo niche ");
    }

}
```

```

public class ChienFou extends Chien {
    @Override
    public void sayHello() {
        System.out.print("chienFou wouf wouf grrr wouf grrr - ");
    }

    public void tournerEnRond() {
        System.out.println(" tourne pas rond dans sa tete ");
    }
}

```

```

public static void operationOnSealed(AnimalDomestique a) {
    if(a instanceof Chien chien) {
        chien.aLaNiche();
        if(chien instanceof ChienFou chienFou) {
            chienFou.tournerEnRond();
        }
    }
    if(a instanceof Chat chat) {
        chat.ronronner();
    }
}

public static void testSealed() {

    var animauxDomestiques = new ArrayList<AnimalDomestique>();
    animauxDomestiques.add(new Chat());
    animauxDomestiques.add(new Chien());
    animauxDomestiques.add(new ChienFou());

    animauxDomestiques.stream().forEach(a-> { a.sayHello(); operationOnSealed(a);});
}

```

affiche

```

chat miaou - ronron
chien wouf wouf - dodo niche
chienFou wouf wouf grrr wouf grrr - dodo niche
tourne pas rond dans sa tete

```

NB : Une interface scellée est implémentable comme un paquets de "record" .

Exemple :

```

public sealed interface Vivant {
    record Vertebre(String name, String p1 , String p2) implements Vivant {}
    record Invertebre(String name, String p3) implements Vivant {}
}

```

NB: Le principal intérêt des classes scellées de java 16/17 tient dans la possibilité d'utiliser un pattern matching au sein d'un switch/case de type d'objet sans partie "default" .

## 6. Pattern matching sur switch/case of object

//NB: still in "preview mode" in java 17-LTS !!!

//java --enable-preview --source 17 ....

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.8.1</version>
  <configuration>
    <release>${java.version}</release>
    <source>${java.version}</source>
    <target>${java.version}</target>
    <compilerArgs>--enable-preview</compilerArgs>
  </configuration>
</plugin>
```

Exemple :

```
package tp.j15_16_17;

public class TestPatternMatchingSwitchPreviewApp {

    static double getDoubleUsingSwitch(Object o) {
        return switch (o) {
            case Integer i -> i.doubleValue();
            case Float f -> f.doubleValue();
            case Double d -> d.doubleValue();
            case String s -> Double.parseDouble(s);
            case null -> 0d;
            default -> 0d;
        };
    }

    static String getTypeAnimalDomesticAsString(AnimalDomestique a) {
        return switch (a) {
            case Chat chat -> "chat";
            case Chien chien -> "chien";
            //default -> "ni chat , ni chien";
        };
    }

    /*
```

*Remarque importante :*

*Ce switch/case avec pattern-matching de type nécessite absolument une partie "default" si AnimalDomestique n'est pas scellé (sans le mot clef sealed et ...)  
et ne nécessite pas de partie "default" si AnimalDomestique est scellé*

-----

Autrement dit l'intérêt principal des classes scellées tient dans la possibilité d'utiliser directement les différents types de classes concrètes (dérivant d'un même type abstrait scellé) au sein d'un switch/case sans default sans avoir besoin de gérer en parallèle une énumération à valeurs possibles fixes/finies.

```

*/
}

static String getTypeVivantFromRecord(Vivant v) {
    return switch (v) {
        case Vivant.Invertebre ive -> "invertebre";
        case Vivant.Vertebre ve -> "vertebre";
        //with default if Vivant is not sealed
    };
}

public static void main(String[] args) {
    System.out.println(getDoubleUsingSwitch("12.5")); //12.5
    System.out.println(getDoubleUsingSwitch(12.6)); //12.6
    System.out.println(getDoubleUsingSwitch(12.7f)); //12.699999809265137
    System.out.println(getDoubleUsingSwitch(12)); //12.0
    System.out.println(getDoubleUsingSwitch(null)); //0.0

    System.out.println(getTypeAnimalDomesticAsString(new Chat())); //chat
    System.out.println(getTypeAnimalDomesticAsString(new Chien())); //chien
    System.out.println(getTypeAnimalDomesticAsString(new ChienFou())); //chien

    System.out.println(getTypeVivantFromRecord(
        new Vivant.Invertebre("limace","pas rapide"))); // invertebre
    System.out.println(getTypeVivantFromRecord(
        new Vivant.Vertebre("homme","intelligent","pas toujours sage"))); // vertebre
    }
}

```

## 7. utilitaire jpackage

L'utilitaire **jpackage** sert à construire des installateurs d'applications pour windows, linux ou mac.

Ça peut construire des fichier ".msi" pour windows.

package\_with\_jpackage.bat

```

cd /d "%~dp0"
REM this script should be run after mvn package or ...

set JAVA_HOME=C:\Program Files\Java\jdk-17
REM NB: on windows10 , with jdk17 , wix tools set is a required dependency of jpackage
REM wix311-binaries.zip can be download from https://github.com/wixtoolset
set WIX_HOME=C:\Prog\wix311
set PATH=%PATH%;%JAVA_HOME%\bin;%WIX_HOME%
REM type=msi or exe for windows , deb or rpm for windows , dmg or pkg on mac
set TYPE=msi
set NAME=JPackagedemoApp

```

```

set MAIN_CLASS=tp.MainClassForJPackageTestApp
set INPUT_JAR_DIR=./target
set MAIN_JAR=tp_java_8_14-0.0.1-SNAPSHOT.jar
set OPTIONS=--win-console --java-options '--enable-preview'

jpackage --input %INPUT_JAR_DIR% --name %NAME% --main-jar %MAIN_JAR%
--main-class %MAIN_CLASS% --type %TYPE% %OPTIONS%
REM fichier construit : JPackageDemoApp.msi

REM apres installation de JPackageDemoApp.msi
REM C:\Program Files\JPackageDemoApp\JPackageDemoApp.exe et runtime java

REM https://www.baeldung.com/java14-jpackage
REM https://www.devdungeon.com/content/use-jpackage-create-native-java-app-installers

```

## 8. Autres apports des versions récentes (12,...,17)

### 8.1. Helpful *NullPointerException* (JEP 358)

Depuis le java 14, les traces générées autour de *NullPointerException* sont beaucoup plus claires/explicites .

**Exemple de code avec bug:**

```

int[] arr = null;
arr[0] = 1;

```

Avec les jdk précédent (ex : v11) :

```

Exception in thread "main" java.lang.NullPointerException
at tp.MyClass.main(MyClass.java:27)

```

Maintenant, la log d'erreur indique plus clairement la source du problème:

```

java.lang.NullPointerException: Cannot store to int array because "arr" is null

```

## 9. Nouveau type de licence depuis java 17

***Le JDK 17 D'Oracle De Nouveau Gratuit Pour Une Utilisation Commerciale***

Le JDK Oracle est à nouveau disponible gratuitement pour une utilisation en production - sous la nouvelle "[Oracle No-Fee Terms et conditions](#)" (NFTC). Cette décision annule une [décision de 2018](#) de facturer l'utilisation en production du JDK d'Oracle et n'affecte pas la [distribution OpenJDK d'Oracle](#). La NFTC s'applique à la version 17 récemment publiée d'Oracle JDK et aux versions futures.

Autre raison de passer à java 17 : java 17 est plus rapide/performant que java 11 .

## XIII - Annexe – détails sur les modules

### 1.1. Descripteur d'un module (module-info.java)

<b>name</b>	nom de notre module (avec d'éventuel(s) "." mais pas de "-" )
<b>dependencies</b> ( <i>requires</i> )	liste des modules dont on dépend
<b>public packages</b> ( <i>exported</i> )	liste des packages que l'on rend accessibles aux autres modules. Normalement, un bon module ne doit exposer publiquement qu'une petite partie de son API (interfaces , pas l'implémentation)
<b>services offered</b> ( <i>provides</i> )	implémentation de services (à injecter/utiliser dans d'autres modules)
<b>services consumed</b> ( <i>uses</i> )	autoriser le module courant à utiliser des services d'autres modules implémentant une certaine interface ou classe abstraite
<b>reflection permissions</b> ( <i>opens</i> )	autoriser ou pas d'autres classes à utiliser la "Reflection java" pour accéder aux éléments privés d'un de nos packages

### 1.2. Variantes de dépendances (requires)

<b>requires</b>	module my.module { requires module.name; }	<b>dépendance à la fois à la compilation et à l'exécution ("runtime")</b>
<b>requires static</b>	module my.module { requires <b>static</b> module.name; }	<b>dépendance à la compilation seulement , facultatif à l'exécution ("runtime")</b>
<b>requires transitive</b>	module my.module { requires <b>transitive</b> module.name; }	<b>dépendance automatique transitive</b> (au niveau du futur module client qui n'aura donc pas besoin d'explicitement les dépendances indirectes)

#### comportement "requires transitive"

Si yy utilise xx utilisant lui même zz et que xx expose une interface de zz  
alors yy est alors obligé d'utiliser zz  
Si le module xx comporte "requires transitive zz"  
alors yy aura juste besoin d'exprimer "requires xx" (pas besoin de requires zz")

#### comportement "requires static"

Le (rare) "requires static moduleOptionnelPasToujoursUtilise"  
signifie "obligatoire à la compilation" et "facultatif à l'exécution"  
avec éventuelle NoClassDefFoundError à gérer au runtime .

### 1.3. Variantes d'exportations / expositions

<b>exports</b>	module my.module { <b>exports</b> com.my.package.name; }	<b>exportation explicite</b>
<b>exports ... to ...</b>	module my.module { <b>export</b> com.my.package.name <b>to</b> com.specific.package; }	<b>(rare) exportation limitée/restreinte à certain(s) autres package(s) ami(s).</b>

#### Comportement des "exports"

NB: **exports p1** n'exporte pas les sous packages **p1.aa** et **p1.bb**

Un même nom complet de package ne peut pas être placé dans plusieurs modules (mais dans un unique module)

### 1.4. interfaces et implémentations de Services

<b>uses</b>	module my.module { <b>uses</b> interfaceOrAbstractClass.name; }	<b>besoin de consommer/injecter un certain type de service</b>
<b>provides ... with ...</b>	module my.module { <b>provides</b> MyInterface <b>with</b> MyImpl; }	<b>Notre module fournit l'implémentation d'un certain type de service</b>

**provides** <type> **with** <type> et **uses** <type> ont été conçus pour mettre en oeuvre une sorte d'injection de dépendances (selon packages d'interfaces) et avec implémentations internes cachées mais cependant chargées au runtime (selon le paramétrage de --module-path )

Exemple de module abstrait (packagé en my\_modules/mod\_aa\_itf.jar) :

```
module tp.module.modaa.itf {
    exports tp.mod.mod_aa.itf;
}
```

et

```
package tp.mod.mod_aa.itf;

public interface MyDisplayApi {
    void display(String msg);
    void displayEx(String msg);
}
```

```
}
```

Exemple de module d'implémentation (packagé en `my_modules/mod_aa_impl.jar`) :

```
module tp.module.modaa.aa.impl1 {
    requires tp.module.modaa.itf;
    provides tp.mod.mod_aa.itf.MyDisplayApi with tp.mod.mod_aa.impl1.MyDisplayImplV1;
}
```

et

```
package tp.mod.mod_aa.impl1;
import tp.mod.mod_aa.itf.MyDisplayApi;

public class MyDisplayImplV1 implements MyDisplayApi {
    public void display(String msg) { System.out.println(">>>" + msg); }
    public void displayEx(String msg) { System.out.println(":::" + msg); }
}
```

Exemple d'utilisation de service :

```
module tp.module.modxx {
    ...
    exports tp.mod.mod_xx_ext;
    requires transitive tp.module.modaa.itf;
    uses tp.mod.mod_aa.itf.MyDisplayApi;
}

// pour uses tp.mod.mod_aa.itf.MyDisplayApi;
// besoin de requires transitive tp.module.modaa.itf;
// mais pas besoin de requires transitive tp.module.modaa.impl1 ou impl2;
```

```
package tp.mod.mod_xx_ext;
import java.util.Iterator;
import java.util.ServiceLoader;
import tp.mod.mod_aa.itf.MyDisplayApi;

public class CxExt {
    private static MyDisplayApi displayService = null;

    public static void displayViaMyDisplayApi(String msg) {
        if(displayService==null) {
            Iterable<MyDisplayApi> myDisplayServices = ServiceLoader.load(MyDisplayApi.class);
            //list of found implementations
            Iterator<MyDisplayApi> myDisplayServicesIterator = myDisplayServices.iterator();
            if(myDisplayServicesIterator.hasNext()) {
```



```

        displayService = myDisplayServicesIterator.next(); //first implementation is choosen
    }
}
if(displayService!=null) {
    displayService.display(msg);
}else{
    System.out.println("no MyDisplayApi implementation found !!!");
}
}
}

```

----> comportement d'un appel à `CxExt.displayViaMyDisplayApi("my top secret message");`

- si aucune implémentation n'est présente dans le `-module-path` ça affiche *"no MyDisplayApi implementation found !!!"*
- si au moins une implémentation (ex : `mod_aa_impl1.jar` ou `mod_aa_impl2.jar`) est présente dans le `-module-path` la première trouvée est alors utilisée et ça affiche *">>> my top secret message"*

## 1.5. Variantes d'ouvertures à l'introspection :

<b>open</b>	<code>open module my.module { }</code>	<i>on autorise tous les autres packages à effectuer de l'introspection sur des éléments privés de l'ensemble des éléments de notre module</i>
<b>opens</b>	<code>module my.module {     opens com.my.package; }</code>	Seul(s) un ou plusieurs(s) de nos packages sont ouvert à de l'introspection avec "accès aux éléments privés"
<b>opens ... to</b>	<code>module my.module {     opens com.my.package to com.specific.package; }</code>	<b>introspection ouverte mais limitée/restreinte à certain(s) autres package(s) ami(s).</b>

**opens** <package> (à ajouter à côté de `exports <package>` ) permet d'autoriser une introspection évoluée (ex: avec `setAccessible(true)` ) .

L'autorisation doit être explicitement donnée (ce qui renforce la sécurité)

Exemple :

Du côté "module ...modxx" :

```

package tp.mod.mod_xx.pub;

public class Cx {
    private String secret="007";
    ...
}

```

et

```

module tp.module.modxx {
    ...
    exports tp.mod.mod_xx.pub;
    opens tp.mod.mod_xx.pub;
}

```

Du côté module client ...modyy :

```

module tp.module.modyy {
    requires tp.module.modxx;
    ...
}

```

et

```

package tp.mod.mod_yy.app;
import java.lang.reflect.Field;
import tp.mod.mod_xx.pub.Cx;

public class MyApp {
    private static void testOpenReflection(Cx objX) {
        Class classX = objX.getClass();
        for(Field f : classX.getDeclaredFields()) {
            f.setAccessible(true);
            String sVal="?";
            try {
                sVal = f.get(objX).toString();
            } catch (IllegalAccessException e) { e.printStackTrace();
                //Exception in thread "main" java.lang.reflect.InaccessibleObjectException:
                //Unable to make field private java.lang.String tp.mod.mod_xx.pub.Cx.secret
                //accessible: module tp.module.modxx does not "opens tp.mod.mod_xx.pub" to module tp.module.modyy
            }
            System.out.println( f.getType().getSimpleName()+ " " + f.getName() + "=" + sVal);
        }
    }

    public static void main(String[] args) {
        Cx objX = new Cx();      testOpenReflection(objX);
    }
}

```

--> affiche une exception de type **java.lang.reflect.InaccessibleObjectException** si **pas** de **opens tp.mod.mod\_xx.pub**; dans ....modxx

--> affiche "**String secret=007**" si **opens tp.mod.mod\_xx.pub**; dans ....modxx

## 1.6. Exemple (partiel) de quelques modules java 9+ :

**NB :** l'aspect "multi-modules maven" n'est pas indispensable mais cela introduit une cohérence structurelle :

**pom.xml** (multi-modules)

```
<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>
  <groupId>tp.mod</groupId>
  <artifactId>mod</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
  <description>modules=nouveautes de java 9 </description>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>11</java.version>
    <maven.compiler.release>${java.version}</maven.compiler.release>
  </properties>

  <modules>
    <module>mod_xx</module>
    <module>mod_yy</module>
  </modules>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.8.0</version>
          <configuration>
            <release>${java.version}</release>
            <source>${java.version}</source> <target>${java.version}</target>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

**module "mod\_xx" (sera utilisé par "mod\_yy")**

**pom.xml**

```
<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>
    <groupId>tp.mod</groupId>
    <artifactId>mod</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <artifactId>mod_xx</artifactId>
  <description>mod_xx java=>9 sera utilisé par mod_yy</description>
  <build>
    <finalName>${project.artifactId}</finalName>
  </build>
</project>

```

### module-info.java (à la racine de src/main/java)

```

module tp.module.modxx {
    exports tp.mod.mod_xx.pub;
    exports tp.mod.mod_xx_ext;
}

```

```

package tp.mod.mod_xx.pub;

import tp.mod.mod_xx.internal.InternalCx;

public class Cx {
    public void fl(String s) {
        InternalCx icx = new InternalCx();
        icx.fli("***" + s);
    }
}

```

```

package tp.mod.mod_xx.internal;

public class InternalCx {
    public void fli(String s) { System.out.println("fl:"+s); }
}

```

```

package tp.mod.mod_xx_ext;

import tp.mod.mod_xx_ext.internal.InternalCxExt;

public class CxExt {
    public static double add(double x, double y) { return InternalCxExt.addition(x, y); }
}

```

```

package tp.mod.mod_xx_ext.internal;

```

```
public class InternalCxExt {
    public static double addition(double x,double y) { return x+y; }
}
```

### module "mod\_yy" (utilisant "mod\_xx")

#### **pom.xml**

```
<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>
        <groupId>tp.mod</groupId>
        <artifactId>mod</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>
    <artifactId>mod_yy</artifactId>
    <description>mod_yy (java=>9) utilisant mod_xx</description>

    <dependencies>
        <dependency>
            <groupId>tp.mod</groupId>
            <artifactId>mod_xx</artifactId>
            <version>0.0.1-SNAPSHOT</version>
        </dependency>
    </dependencies>

    <build>
        <finalName>${project.artifactId}</finalName>
    </build>
</project>
```

#### **module-info.java** (à la racine de src/main/java)

```
module tp.module.modyy {
    requires tp.module.modxx;
    requires java.desktop;
    exports tp.mod.mod_yy.pub;
}
```

```
package tp.mod.mod_yy.pub;
import tp.mod.mod_yy.internal.InternalCy;
```

```
public class Cy {
    public void f2(String s) {
        InternalCy icy = new InternalCy();
        icy.f2i("##" + s);
    }
}
```

```
}
}
```

```
package tp.mod.mod_yy.internal;
//import tp.mod.mod_xx.internal.InternalCx; //pas accessible car package pas exporté
//import tp.mod.mod_xx_ext.internal.InternalCxExt; //pas accessible car package pas exporté

import tp.mod.mod_xx.pub.Cx;
import tp.mod.mod_xx_ext.CxExt;

public class InternalCy {
    public void f2i(String s) { System.out.println("f2:"+s); }
    public void f1i(String s) { Cx cx = new Cx(); cx.f1(s); }
    public static double ajouter(double a,double b) { return CxExt.add(a, b); }
    //code impossible (basé sur classe InternalCx pas accessible / pas exportée):
    //public void forbidden_f1(String s) { InternalCx icx = new InternalCx(); icx.f1i(s); }
}
```

```
package tp.mod.mod_yy.app;
import javax.swing.JOptionPane;
import tp.mod.mod_yy.internal.InternalCy;
public class MyApp {
    public static void main(String[] args) {
        InternalCy icy = new InternalCy();
        icy.f2i("java");icy.f1i("java");
        JOptionPane.showMessageDialog(null, "ok with requires java.desktop");
    }
}
```

## 1.7. Modules explicites , automatiques et "unnamed"

### Modules explicites gérés rigoureusement :

#### accès contrôlés si `module.info.java` est présent

Si dans le module `mod_yy` (utilisant `mod_xx`) , le fichier **`module.info.java`** est présent  
il faut alors tout rendre cohérent (exports , requires) :  
et alors **seuls les éléments bien exportés par `mod_xx`**  
**et bien importés (via requires) dans `mod_yy` seront accessibles dans `mod_yy`.**

### Modules automatiques sans gestion rigoureuse :

#### accès libres **mais pas rigoureux** si pas de `module.info.java` (modules "automatiques")

NB: si jamais de `module.info.java` (pour chaque module)  
alors tout ce qui est publique est accessible --> comme ancien comportement de java <=8 .

#### **requires \* par défaut si `module.info.java` est absent (du coté module utilisateur)**

Si le module `mod_xx` comporte `module.info.java` (avec certains éléments exportés),  
alors le module `mod_yy` (utilisant `mod_xx`) pourra alors éventuellement ne pas comporter de  
fichier `module.info.info` et dans ce cas le comportement semble équivalent à  
**requires \* (vérifié)**  
*et (à priori) exports \**

NB : On appel **module automatique** un *.jar ne comportement pas de fichier `module-info.class`* et pourtant **placé** dans un des répertoires du `--module-path` .

Un tel module se voit alors attribué un nom automatiquement basé sur le nom du .jar et quelques ajustements (ex : `libs-legacy/xxx-yyy-legacy-1.0-SNAPSHOT.jar` est automatiquement nommé `xxx.yyy.legacy` ) et un *requires xxx.yyy.legacy* est donc envisageable depuis un autre module .

Les modules automatiques voient tous leurs types publics exportés ("*exports \**" automatique). Un module utilisant un module automatique peut donc utiliser n'importe quel type public

Attention, maven et eclipse ne gèrent pas très bien les modules automatiques lors de l'exécution d'une application java.

Modules "**unnamed**" pour les .jar chargés depuis le `--class-path` plutôt que `--module-path`:

NB : On appelle **module "unnamed"** le méga module constitué de tous les **.jar** qui ne sont pas chargés via le `--module-path` mais qui sont (à l'ancienne) chargés via l'option `--class-path` ayant `-classpath` et `-cp` comme *alias* .

Cet "unnamed module" a la particularité d'avoir accès à :

- tous les packages exportés par tous les autres modules disponibles dans le module-path
- tous les jars du classpath (ie: tous les autres types présents dans cet unnamed module)

Le module par défaut (sans nom) "unnamed module" exporte tous ses packages.

Cependant les classes de cet "unnamed module" ne sont accessibles que depuis ce même "unnamed module".

NB : Par défaut, aucun "named module" ne peut accéder aux classes du "unnamed module".

*(that was done on purpose to prevent modular JARs from depending on "the chaos of the class path". )*

Eventuel ajout de modules (chargés via `--module-path` ) dans la sphere de visibilité du "unnamed module"

Il est possible d'ajouter un fichier ".jar" (hors jdk récent mais chargé via le `--module-path`) au "unnamed / default module" via l'option de `--add-modules` de java :

Exemple :

```
java .... --add-modules java.xml.bind
```

ou bien(via maven) :

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.8.1</version>
  <configuration>
    <source>11</source><target>11</target>
    <release>11</release>
    <compilerArgs>
      <arg>--add-modules</arg>
      <arg>java.xml.bind</arg>
    </compilerArgs>
  </configuration>
</plugin>
```

Ceci est essentiellement utile pour l'ancien l'écosystème JEE .



## **1.8. Options (très pointues) disponibles en ligne de commande pour ré-ajuster la configuration des modules**

**--add-reads** <module>=<target-module>(,<target-module>)\*

met à jour <module> pour lire <target-module>, sans tenir compte de la déclaration de module.

<target-module> peut être ALL-UNNAMED pour lire tous les modules sans nom.

**--add-exports** <module>/<package>=<target-module>(,<target-module>)\*

met à jour <module> pour exporter <package> vers <target-module>, sans tenir compte de la déclaration de module. <target-module> peut être ALL-UNNAMED pour effectuer un export vers tous les modules sans nom.

**--add-opens** <module>/<package>=<target-module>(,<target-module>)\*

met à jour <module> pour ouvrir <package> vers <target-module>, sans tenir compte de la déclaration de module

**--patch-module** <module>=<file>(;<file>)\*

Remplacement ou augmentation d'un module avec des classes et des ressources dans des fichiers ou des répertoires JAR.

# XIV - Annexe – Quelques API de java

## 1. Process Api

### 1.1. Lancement d'un processus depuis une application java

```
public static void startNodePad() {
    ProcessBuilder builder = new ProcessBuilder("notepad.exe"); //de java.lang depuis java 1.5
    try {
        Process process = builder.start();
        System.out.println("pid of process=" + process.pid());
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

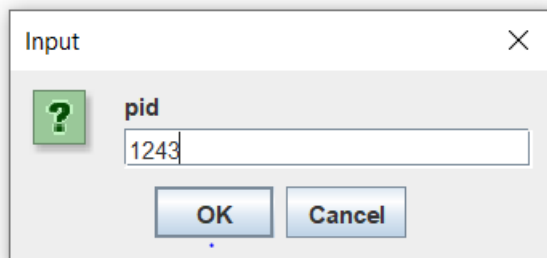
### 1.2. L'interface ProcessHandle (java.lang) depuis java 9

return type	method	fonctionnalités
static <a href="#">Stream</a> < <a href="#">ProcessHandle</a> >	<a href="#"><u>allProcesses()</u></a>	Returns a snapshot of all processes visible to the current process.
<a href="#">Stream</a> < <a href="#">ProcessHandle</a> >	<a href="#"><u>children()</u></a>	Returns a snapshot of the current direct children of the process.
int	<a href="#"><u>compareTo</u></a> ( <a href="#">ProcessHandle</a> other)	Compares this ProcessHandle with the specified ProcessHandle for order.
static <a href="#">ProcessHandle</a>	<a href="#"><u>current()</u></a>	Returns a ProcessHandle for the current process.
<a href="#">Stream</a> < <a href="#">ProcessHandle</a> >	<a href="#"><u>descendants()</u></a>	Returns a snapshot of the descendants of the process.
boolean	<a href="#"><u>destroy()</u></a>	Requests the process to be killed.
boolean	<a href="#"><u>destroyForcibly()</u></a>	Requests the process to be killed forcibly.
boolean	<a href="#"><u>equals</u></a> ( <a href="#">Object</a> other)	Returns true if other object is non-null, is of the same implementation, and represents the same system process; otherwise it returns false.
int	<a href="#"><u>hashCode()</u></a>	Returns a hash code value for this ProcessHandle.
<a href="#">ProcessHandle.Info</a>	<a href="#"><u>info()</u></a>	Returns a snapshot of information about the process.
boolean	<a href="#"><u>isAlive()</u></a>	Tests whether the process represented by this ProcessHandle is alive.
static <a href="#">Optional</a> < <a href="#">ProcessHandle</a> >	<a href="#"><u>of</u></a> (long pid)	Returns an <a href="#">Optional</a> < <a href="#">ProcessHandle</a> > for an existing native process.

<a href="#">CompletableFuture&lt;ProcessHandle&gt;</a>	<a href="#">onExit()</a>	Returns a <a href="#">CompletableFuture&lt;ProcessHandle&gt;</a> for the termination of the process.
<a href="#">Optional&lt;ProcessHandle&gt;</a>	<a href="#">parent()</a>	Returns an <a href="#">Optional&lt;ProcessHandle&gt;</a> for the parent process.
long	<a href="#">pid()</a>	Returns the native process ID of the process.
boolean	<a href="#">supportsNormalTermination()</a>	Returns true if the implementation of <a href="#">destroy()</a> normally terminates the process.

### 1.3. Exemples de gestion de processus

```
import java.io.IOException;
import java.util.Optional;
import javax.swing.JOptionPane;
....
public static void destroyProcessById() {
    long pid=Long.parseLong(JOptionPane.showInputDialog(null, "pid"));
    Optional<ProcessHandle> optionalProcessHandle = ProcessHandle.of(pid);
    optionalProcessHandle.ifPresent(processHandle -> processHandle.destroy());
}
```



```
public static void startAndDestroyProcess() {
    //ProcessBuilder builder = new ProcessBuilder("C:\\Program Files\\Mozilla Firefox\\firefox.exe");
    ProcessBuilder builder = new ProcessBuilder("notepad.exe");
    try {
        Process process = builder.start();
        System.out.println("pid of process=" + process.pid());
        Thread.sleep(5000); //5000ms
        process.destroy();
        if (process.isAlive()) {
            System.out.println("process still alive ...");
            process.destroyForcibly(); //a utiliser que si process.destroy() ne suffit pas
        }else {
            System.out.println("process is no more alive ...");
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
}
}
```

```
...
public static String processHandleAsString(ProcessHandle ph) {
    return "pid="+ph.pid()+" - "+ph.info().toString();
}

public static void displayingAllProcess() {
    ProcessHandle.allProcesses().forEach(ph -> System.out.println(processHandleAsString(ph)));
}
-->
```

```
...
pid=21964 - [user: Optional[LAPTOP-DDC\didier], cmd: C:\Program Files (x86)\Google\Chrome\Application\chrome.exe,
startTime: Optional[2020-07-02T14:04:20.296Z], totalTime: Optional[PT0.078125S]]
pid=4840 - []
pid=5696 - [user: Optional[LAPTOP-DDC\didier], cmd: C:\Program Files\Java\jdk-11.0.4\bin\javaw.exe, startTime:
Optional[2020-07-02T14:47:38.376Z], totalTime: Optional[PT0.34375S]]
```

```
package org.mycontrib.tp;
public class MySubProcess {
    public static void main(String[] args) {
        System.out.println("MySubProcess ...");
        try { Thread.sleep(5000); /*5000ms*/ } catch (InterruptedException e) { e.printStackTrace(); }
        System.exit(0);
    }
}
```

```
public static void startAndWaitingProcess() {
    ProcessBuilder builder = new ProcessBuilder();
    /*
    final String javaHome = "C:\\Program Files\\Java\\jdk-11.0.4"; // " " need around directory name with space
    final String javaCmdeWithQuote = "\"" + javaHome + "\\bin\\java.exe" + "\"";
    //final String cmde = javaCmdeWithQuote + " -version";
    final String cmde = javaCmdeWithQuote + "-cp .\\target\\myProcessApp.jar org.mycontrib.tp.MySubProcess";
    System.out.println("cmde="+cmde);
    //builder.command("cmd.exe", "/c", cmde); // ok, "/c" = terminate after this run
    */
    //NB: .\\target\\myProcessApp.jar will be ready after mvn install or ...
    builder.command("java", "-cp", ".\\target\\myProcessApp.jar",
        "org.mycontrib.tp.MySubProcess"); //ok with java in PATH
    boolean isStopped = false;
    try {
        Process process = builder.start();
        System.out.println("pid of process=" + process.pid());
        BufferedReader outputReaderOfSubProcess = new BufferedReader(new
            InputStreamReader(process.getInputStream()));
        System.out.println("output of process (first line) =" +
            outputReaderOfSubProcess.readLine());
        isStopped=process.waitFor(3, TimeUnit.SECONDS);
        if(isStopped) {
```

```
        System.out.println("process is terminated");
    } else {
        System.out.println("process not terminated after 3s");
        isStopped=process.waitFor(3, TimeUnit.SECONDS);//re-wait
    }
    if (isStopped) {
        System.out.println("process is stopped with exit value="
            +process.exitValue());
    }
} catch (Exception e) {
    e.printStackTrace();
}
}
```

## XV - Annexe – Tests unitaires (JUnit 3 , 4 et 5)

### 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* . JUnit existe en versions 3 et 4.

La **version 4** utilise des **annotations** pour son paramétrage (**@Test** , **@Before** , ....)

#### 1.2. Structure d'une classe de test (ancienne version JUnit3)

##### (Ancien) JUnit 3

héritage

**Conventions  
de noms sur  
méthodes**  
**setUp()**,  
**testXy()**  
et  
**tearDown()**

```
import junit.framework.TestCase;

public class CalculateurTest extends TestCase {
    private Calculateur c; //objet/composant à tester

    protected void setUp(){ //initialisation du composant à tester
        c = new Calculateur(); //setUp() appelée avant chaque testXy()
    }

    public void testAdd() {
        assertEquals( c.add(5,6) , 11 , 0.000001 );
        // ou assertTrue(condition_a_vérifier) .
    }

    public void testMult() {
        assertEquals( c.mult(5,6) , 30 , 0.000001 );
    }

    protected void tearDown(){
        // éventuel code de terminaison réinitialisant certaines valeurs
        // d'un jeu de données (méthode facultative) }
    }
}
```

JUnit 3 est basée sur des conventions de nommage:

- La méthode **setUp()** sera appelée automatiquement pour initialiser les valeurs de certains objets qui seront ultérieurement utilisés au sein des tests.
- 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).

### 1.3. Structure d'une classe de test ( version actuelle JUnit4 )

#### **JUnit4** (avec annotations)

Plus besoin  
d'hériter de  
TestCase  
mais  
**@Before**  
**@After**  
et  
**@Test**  
attendus

```
import org.junit.Assert;
import org.junit.Test; import org.junit.Before;

public class CalculateurTest
{ private Calculateur c;

  @Before /* comportement proche d'un constructeur par défaut */
  public void initialisation(){
    c = new Calculateur(); // déclenché avant chaque @Test .
  }

  @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 );
  }
}
```

*Méthode statique*

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 ???) \* ...

## 1.4. Fonctionnement de JUnit

### Une instance de "Test JUnit" pour chaque test unitaire !!!

**La technologie JUnit (en version 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 !!!!

→ la notion d'ordre d'appel des méthodes est inexistante (non applicable) sur une classe de test JUnit.

Ceci permet d'obtenir des tests unitaires complètement indépendants mais ceci peut quelquefois engendrer certaines lenteurs ou lourdeurs.

Certains contextes (plutôt "stateless") peuvent se prêter à **des optimisations "static"** (**@BeforeClass** , **@AfterClass**) .

En combinant JUnit4 avec d'autres technologies (ex : SpringTest) , on peut également effectuer quelques optimisations (initialisations spéciales) au cas par cas selon le(s) framework(s) utilisé(s).

**Assert.assertNotNull()** et **Assert.fail("message")** peuvent être pratiques dans certains cas (exceptions non remontées, ...)

### Ordre des méthodes de test sur une classe JUnit4

Une classe de test **JUnit4** peut comporter plusieurs méthodes de test. Chacune de ces méthodes correspond à un **test unitaire** (censé être indépendant des autres) et donc par défaut , pour garantir une bonne isolation entre les différents tests unitaires :

- l'ordre des méthodes de tests déclenchées est non déterministe
- chaque méthode de test est exécutée avec une instance différente de la classe de test

**Dans certains cas rares et pointus**, on peut préférer **contrôler l'ordre des méthodes qui seront appelées**. Les tests seront alors un peu **moins "unitaires"** et un peu plus "liés".

Ceci peut s'effectuer de deux manières :

- \* avec l'annotation **@FixMethodOrder** que l'on trouve sur les versions récentes de JUnit4 .
- \* En écrivant une classe contrôlant le déclenchement d'une **série ordonnée de tests unitaires** (avec **@Suite**) .



1.5. **@BeforeClass** et "static" (optimisations)**JUnit4** (avec "*static*" et "*@BeforeClass*")**@BeforeClass****@AfterClass**

attendus pour

gérer des

éléments

"static"

```

import org.junit.Assert;
import org.junit.Test; import org.junit.Before;

public class CalculateurTest
{ private static Calculateur c;

  @BeforeClass /* appelée une seule fois */
  public static void initialisation(){
    c = new Calculateur(); // initialisation "static".
  }

  @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 );
  }
}

```

*Méthode statique*

## 1.6. Suite de tests (@Suite)

### Suite ordonnées de tests (JUnit4)

Quelques usages potentiels (tests ordonnés):

séquence Create/insert , select , update , select , delete , ...

variantes au niveau intégration continue :

- Suite\_tests\_essentiel (pour build rapides)
- Suite\_complete\_tests (pour build de nuit)

*exemple :*

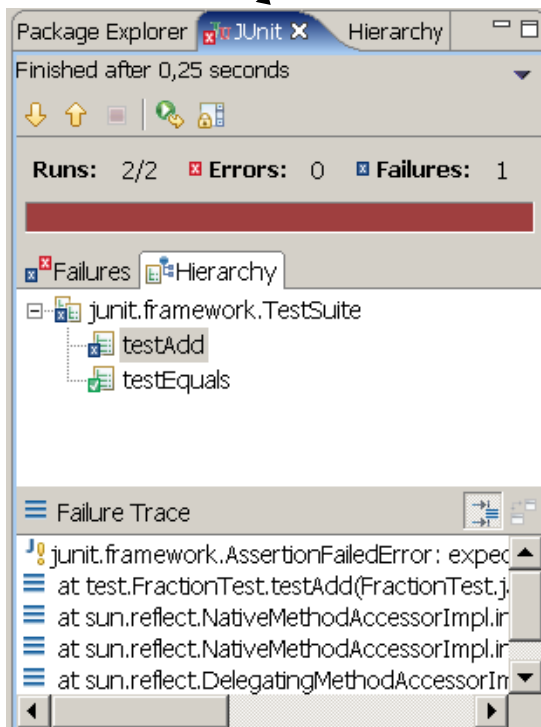
```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({
    JunitTest1.class,
    JunitTest2.class
})
public class JunitTestSuite {
}
```

## 1.7. Lancement des tests unitaires

### Lancement des tests unitaires

Depuis l'IDE eclipse:

**Run as ... / JUnit test**



TestSuite en JUnit3

et lancement après une sélection de **package** en JUnit4 pour lancer d'un coup toute une série de tests.

Comptabilisations :

**Error(s) :** exceptions java  
non rattrapées.

**Failure(s) :** assertions  
non vérifiées.

**VERT si aucune erreur.**

Depuis "maven" :

coder des classes nommées "**Test**Xy"  
ou "**XYTest**" dans **src/test/java**  
et lancement via **mvn test** ou autre.

## Combinaisons de frameworks (de tests) via **@RunWith**

```
import org.junit.runner.RunWith;  
import org.springframework.test.context.ContextConfiguration;  
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;  
...  
// nécessite spring-test.jar et junit4.11.jar dans le classpath  
@RunWith(SpringJUnit4ClassRunner.class)  
// Chargement automatique de "/mySpringConf.xml" pour la configuration  
@ContextConfiguration(locations={"/mySpringConf.xml"})  
public class TestGestionComptes {  
  
// injection/initialisation du composant à tester contrôlée par @Autowired (de Spring)  
@Autowired //ou bien @Inject  
private InterfaceServiceXY serviceXy = null;  
  
@Test  
public void testTransferer(){  
    serviceXy.transferer(...); Assert.assertTrue( ... );  
    }  
}
```

Il existe aussi **@RunWith(MockitoJUnitRunner.class)** et autres.

## 2. Tests unitaires avec JUnit 5

### 2.1. Présentation de JUnit 3,4,5

**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 des IDE Eclipse et IntelliJ.*

JUnit existe en versions 3 , 4 et 5 avec des différences significatives d'une version à l'autre .

### 2.2. Présentation des anciennes versions 3 et 4

La très ancienne version 3 de JUnit n'utilisait pas d'annotations . Tout était basé sur un héritage (TestCase) et sur des conventions de nom sur les méthodes (setUp() , tearDown() , testXy() ) . Depuis environ 2006/2007 , cette historique version 3 est devenue petit à petit obsolète (utilisée seulement dans les anciens projets).

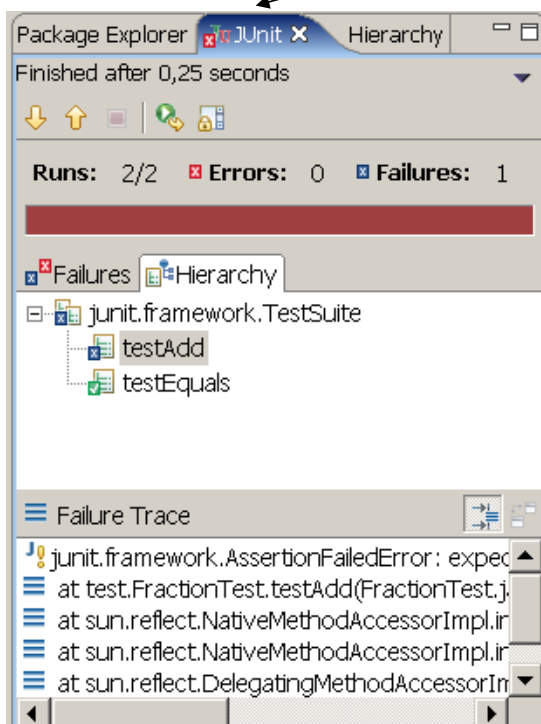
La version 4 a été massivement utilisée dans la majorité des projets java de 2007 à 2019 environ . La version 4 était basée sur le package org.junit (Test, Assert, ...) et les principales annotations étaient @Test , @Before , @After , @BeforeClass , @AfterClass .

### 2.3. Lancement des tests unitaires

#### Lancement des tests unitaires

Depuis l'IDE eclipse:

**Run as ... / JUnit test**



TestSuite en JUnit3

et lancement après une sélection de **package** en JUnit4 pour lancer d'un coup toute une série de tests.

Comptabilisations :

**Error(s) :** exceptions java non rattrapées.

**Failure(s) :** assertions non vérifiées.

**VERT si aucune erreur.**

Depuis "maven" :

coder des classes nommées "**TestXy**" ou "**XYTest**" dans **src/test/java** et lancement via **mvn test** ou autre.

## 2.4. Présentation de JUnit 5 ( "jupiter" )

La version 5 de JUnit a été entièrement restructurée et s'appuie sur certaines nouvelles fonctionnalités apportées par la version 8 du langage java (lambda expressions, ....) .

### Principales différences entre JUnit4 et JUnit5 :

JUnit 4	JUnit 5
package org.junit	package <b>org.junit.jupiter.api</b>
Assert.assertTrue() , Assert.assertEquals(...)	<b>Assertions</b> .assertTrue() , <b>Assertions</b> .assertEquals(...)
@Before , @After	<b>@BeforeEach</b> , <b>@AfterEach</b>
@BeforeClass , @AfterClass (avec static)	<b>@BeforeAll</b> , <b>@AfterAll</b> (avec static)
...	...

## 2.5. Configuration maven pour JUnit 5 ( "jupiter" )

```

...
<properties>
  <junit.jupiter.version>5.4.2</junit.jupiter.version>
</properties>
<dependencies>
<dependency>
  <groupId>org.junit.jupiter</groupId> <artifactId>junit-jupiter-api</artifactId>
  <version>${junit.jupiter.version}</version>   <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId> <artifactId>junit-jupiter-engine</artifactId>
  <version>${junit.jupiter.version}</version>   <scope>test</scope>
</dependency>
</dependencies>
<build> <plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0-M4</version> <!-- Need at least 2.22.0 to support JUnit 5 -->
  </plugin>
</plugins> </build> ...

```

Eventuels compléments :

- junit-jupiter-params**      *support des **tests paramétrés** avec JUnit Jupiter.*
- junit-vintage-engine**      *pour interpréter et exécuter des anciens tests codés en JUnit 3 ou 4*
- junit-platform-....**      *pour intégration et lancement (console , maven , gradle , ....)*

## 2.6. Test Unitaire avec JUnit 5 (@Test, @BeforeEach, @AfterEach)

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
...
public class TestCalculatrice {
    private static Logger logger = LoggerFactory.getLogger(TestCalculatrice.class);
    private CalculatriceEx calculatriceEx; //à tester

    @BeforeEach // @BeforeEach in jUnit5 , @Before in jUnit4
    public void initCalculatriceEx() {
        this.calculatriceEx = new CalculatriceEx();
    }

    @Test
    public void testSum() {
        calculatriceEx.pushVal(1.1);
        calculatriceEx.pushVal(2.2);
        calculatriceEx.pushVal(3.3);
        double somme = calculatriceEx.sum();
        logger.trace("somme(1.1, 2.2, 3.3)="+somme);
        //Assert.assert...() avec jUNit 4 et Assertions.assert...() avec jUnit5/jupiter
        Assertions.assertEquals( 6.6 , somme, 0.00000001);
    }

    @Test
    public void testAverage() {
        calculatriceEx.pushVal(1.1);
        calculatriceEx.pushVal(1.5);
        double moyenne = calculatriceEx.average();
        logger.trace("moyenne(1.1, 1.5)="+moyenne);
        Assertions.assertEquals( 1.3 , moyenne, 0.00000001);
    }
}
```

**NB:** Comme avec JUnit 4, **par défaut JUnit 5 crée une nouvelle instance de la classe de test pour exécuter chaque méthode de test. (@TestInstance(Lifecycle.PER\_METHOD) par défaut)**

Assertions.**assertEquals**(*expectedValue*, *effectiveValue* , *delta* ) ;  
 Assertions.**assertTrue**(booleanExpression) ;    Assertions.**notNull**(...) ...

NB :

```
import static org.junit.jupiter.api.Assertions.assertTrue;
```

permet d'écrire **assertTrue** (res==5) au lieu de *Assertions.assertTrue*(res==5)  
et ça peut aider à masquer la différence de préfixe entre JUnit 4 et JUnit5 .

## 2.7. Test Unitaire JUnit5 avec static (@BeforeAll , @BeforeAll )

```
...
import org.junit.jupiter.api.BeforeAll;
...
public class TestCalculatrice {

    private static SimpleCalculatrice calculatrice; //à tester

    @BeforeAll //@BeforeAll in jUnit5 , @BeforeClass in jUnit4
    public static void initCalculatrice() {
        calculatrice = new SimpleCalculatrice();
    }

    @Test
    public void testAddition() {
        double resAdd=calculatrice.addition(5.5, 6.6);
        logger.trace("addition(5.5,6.6)="+resAdd);
        Assertions.assertTrue(resAdd>= (12.1 - 0.00000001) &&
                               resAdd <= (12.1 + 0.00000001));
    }

    @Test
    public void testMultiplication() {
        double resMult=calculatrice.multiplication(2.0, 3.3);
        logger.trace("multiplication(2.0, 3.3)="+resMult);
        Assertions.assertEquals( 6.6 , resMult, 0.00000001);
    }
}
```

## 2.8. Spécificités de JUnit5

```
@DisplayName("Ma classe de test JUnit5 que j'aime")
public class MyTest {

    @Test
    @DisplayName("Mon cas de test Xy qui va bien")
    void Testxy() {
        // ...
    }
}
```

NB: Les valeurs des @DisplayName seront affichées au sein des rapports



d'exécution des tests

**NB** : contrairement à JUnit4, les **méthodes** préfixées par @BeforeEach , @Test , .... **n'ont plus absolument besoin d'être public** (protected ou bien **la visibilité implicite par défaut de niveau package suffit** ) .

AssertAll() avec lambdas :

```
Assertions.assertAll("wrong Dimension",
    () -> Assertions.assertTrue(elt.getWidth() == 400, "wrong width"),
    () -> Assertions.assertTrue(elt.getHeight() == 500, "wrong height"));
```

vérifie que chacune des lambdas est ok (sans exception) .

```
Iterable<Integer> attendu = new ArrayList<>(Arrays.asList(1, 2, 3));
Iterable<Integer> actuel = new ArrayList<>(Arrays.asList(1, 2)); //manque 3
Assertions.assertIterableEquals(attendu, actuel); //echec
```

vérifie que les collections ont mêmes tailles et mêmes valeurs.

**assertEquals()** vérifie mêmes valeurs

**assertSame()** vérifie références sur même instance/objet .

AssertThrows avec lambda :

```
@Test
void verifierException() {
    String valeur = "123a";
    Assertions.assertThrows(NumberFormatException.class, () -> {
        Integer.valueOf(valeur);
    });
}
```

vérifie qu'une exception est bien levée (suite à erreur volontaire) et du bon type .

AssertTimeout avec lambda :

```
@Test
void verifierTimeout() {
    Assertions.assertTimeout(Duration.ofMillis(200), () -> {
        * traitement potentiellement trop lent *
        return "...";
    });
}
```

**@Disabled**("test temporairement désactivé")  
sur classe ou méthode de test

**@RepeatedTest**(value = 3) est une variante de @Test permettant de lancer n fois un même test

## 2.9. Suppositions (pour exécuter ou pas un test selon le contexte) :

```
assumeTrue( System.getenv("OS").startsWith("Windows") );
```

//la suite du test ne sera exécutée que si os courant est Windows

//si le test n'est pas exécuté --> même comportement que si test vide --> ok/réussi/vert

*//variante avec lambda exécutée qui si supposition ok :*

```
assumingThat(System.getenv("OS").startsWith("Windows"), () -> {  
    assertTrue(new File("C:/Windows").exists(), "Repertoire Windows inexistant");  
});
```

## 2.10. Tests imbriqués de JUnit5

```
public class MyTest {  
  
    @BeforeEach  
    void mainInit() {  
        System.out.println("BeforeEach / first level");  
    }  
  
    @Nested  
    class MonTestImbrique {  
        @BeforeEach  
        void subInit() {  
            System.out.println("BeforeEach imbrique");  
            valeur = 5;  
        }  
  
        @Test  
        void simpleTestImbrique() {  
            System.out.println("SimpleTest imbrique valeur=" + valeur);  
            Assertions.assertEquals(5, valeur);  
        }  
    }  
}
```

NB : En plaçant **@TestInstance(Lifecycle.PER\_CLASS)** au dessus d'une classe de test JUnit5 (imbriquée ou pas) , on a comme comportement le fait qu'une seule instance de la classe de test sera utilisée pour exécuter successivement toutes les méthodes de tests de la classe de test (contrairement au comportement par défaut PER\_METHOD) .

NB2 : Le mode PER\_CLASS permet aussi :

- que les méthodes annotées avec @BeforeAll et @AfterAll n'aient pas l'obligation d'être statiques
- d'utiliser les annotations @BeforeAll et @AfterAll sur des méthodes de classes annotées avec @Nested

## 2.11. Tests paramétrés via source/série de valeurs

Après avoir ajouté la dépendance nécessaire *junit-jupiter-params* , on peut écrire des méthodes de tests avec un paramètre qui sera alimenté avec une source/série de valeurs .

Un test paramétré sera ainsi lancé plusieurs fois (avec des arguments différents pour obtenir une certaine variété)

### @ParameterizedTest

```
@ValueSource(ints = { 1, 2, 3 })
void testParametreAvecValueSource(int valeur) {
    assertEquals(valeur + valeur, valeur * 2);
}
```

@ValueSource(ints = { 1, 2, 3 }) ou @ValueSource(strings = { "un", "deux" }) ou ...

il existe également @EnumSource , @MethodSource , @CsvSource , ....

import java.util.stream.Stream;

```
@ParameterizedTest
@MethodSource("fournirDonneesParametres")
void testTraiterSelonMethodSource(String element) {
    assertTrue(element.startsWith("elt"));
}

static Stream<String> fournirDonneesParametres() {
    return Stream.of("elt1", "elt2");
}
```

## 2.12. Tests dynamiques / @TestFactory

```
...
import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.ArrayList;
```

```
import java.util.Collection;
import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;

class MonTestSimple {

    @TestFactory
    Collection<DynamicTest> dynamicTestsWithCollection() {
        Collection<DynamicTest> myDynamicTests = new ArrayList<>();
        for (int i = 1; i <= 5; i++) {
            int val = i;
            myDynamicTests.add(DynamicTest.dynamicTest("Ajout " + val + "+" + val,
                () -> assertEquals(val * 2, val + val)));
        }
        return myDynamicTests;
    }
}
```

--> à priori intéressant que si couplé avec une introspection dynamique (framework de tests).

## 2.13. Suite de Tests et Tags

**@Tag("nomDeTag")** de JUnit5 peut être placé sur une classe ou bien une méthode de test

Ces noms de tags (préalablement appelés catégories en Junit4) permettront d'effectuer ultérieurement des filtrages sur les tests à lancer ou pas .

Il est éventuellement possible de placer plusieurs tags complémentaires au dessus d'un même test (ex : **@Tag("prod") @Tag("dev")** ) .

```
import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;
```

```
@RunWith(JUnitPlatform.class)
@SelectPackages({ "com.xyz.test.p1" })//ne lancer que les tests de ce(s) package(s)
@IncludeTags({ "prod", "dev" }) //ne lancer que les tests qui ont ces tags
//il existe aussi @ExcludeTags ne pouvant pas être utilisé en même temps que @IncludeTags
public class MyTestSuite1
{
}
}
```

NB : par défaut , **@SelectPackages(...)** sélectionne tous les sous-packages également .

On peut éventuellement ajouter **@IncludePackages(...)** ou bien **@ExcludePackages(...)** pour filtrer les sous-packages à sélectionner.

On peut également ajouter **@IncludeClassNamePatterns({ "^.\*Simple\$" })** pour filtrer les noms des classes de Tests à lancer .

# XVI - 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 annotations 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 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
<b>@Deprecated</b>	Éléments (méthode , attribut, ...) devenu obsolète et qui ne devrait plus être utilisé. <b>NB:</b> l'utilisation conjointe du <i>commentaire javadoc</i> <code>@deprecated</code> permet en plus d'indiquer la raison et une suggestion de remplacement: /** * <i>@deprecated pas bien car ... , utiliser .... à la place</i> **/ <b>@Deprecated</b>
<b>@Override</b>	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.
<b>@SuppressWarnings</b>	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: <b>@SuppressWarnings({"deprecation","unchecked"})</b>

### 2.2. Méta-annotations (Annotations sur annotations ):

Nécessite =====> `import java.lang.annotation.*;`

exemple:

**@Documented**

`public interface @MyDocumentedAnnotation { ...}`

Méta-Annotations du jdk 1.5	Significations
<b>@Documented</b>	L'annotation doit apparaître dans la documentation générée par javadoc ou ...
<b>@Inherit</b>	L'annotation sera automatiquement héritée par les sous classes ( <b>@Inherit</b> ne peut être utilisée que sur annotation de classe – ce n'est pas utilisable sur une annotation d'interface).
<b>@Retention(...)</b>	Pour indiquer la portée (ou durée de vie) de l'annotation. <b>RetentionPolicy.SOURCE</b> (code source uniquement) <b>RetentionPolicy.CLASS</b> (dans .class également mais pas pris en compte par la JVM , c'est la valeur par défaut). <b>RetentionPolicy.RUNTIME</b> ( vu par la JVM et l'api "reflection" ).
<b>@Target(...)</b>	Pour que l'annotation ne soit utilisable que devant une ou plusieurs

Méta-Annotations du jdk 1.5	Significations
	<p>catégories d'éléments (ex: classe , méthode , ....).</p> <p>Si une annotation n'est pas bien utilisée (erreur de @Target) ==&gt; Erreur du compilateur.</p> <p>Si la méta-annotation @Target n'est pas précisée , l'annotation peut être utilisée sur n'importe quel élément .</p> <p><b>ElementType.ANNOTATION_TYPE</b> (devant annotation)</p> <p><b>ElementType.CONSTRUCTOR</b> , <b>ElementType.FIELD</b> ,</p> <p><b>ElementType.LOCAL_VARIABLE</b> ,</p> <p><b>ElementType.METHOD</b> , <b>ElementType.PACKAGE</b></p> <p><b>ElementType.PARAMETER</b></p> <p><b>ElementType.TYPE</b> (devant classe , interface ou enum).</p>

### 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;
    /** Énumération 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.apr.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) {
        Messenger m = env.getMessenger();
        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 Réflexion (`java.lang.reflect`) a été étendue en version 1.5 de façon à supporter les annotations. Pour cela, les classes **Package**, **Class**, **Constructor**, **Method** et **Field** possèdent quatre nouvelles méthodes décrites 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();

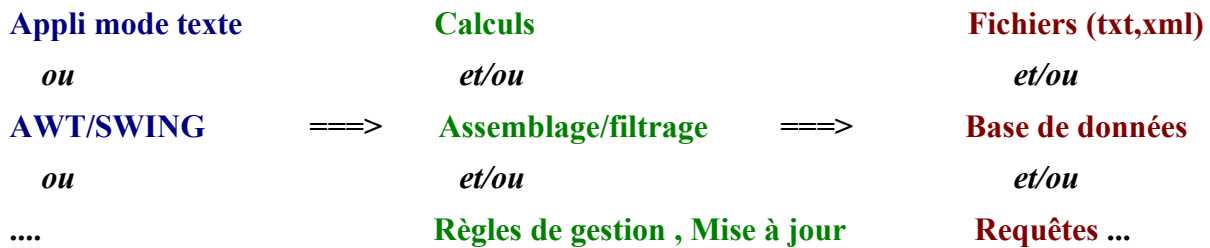
// On regarde si la classe possède une annotation :
MonAnnotation annotation = classInstance.getAnnotation(MonAnnotation.class);

if (annotation!=null) {
    System.out.println ("MonAnnotation : " + annotation.value() );
}
```

# XVII - Annexe – Structure globale appli.

## 1. Architecture généralement recommandée

Partie "Présentation" ==> Partie "Traitements métiers" ==> Partie "Accès aux données"



### 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 – 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 **getXxx()/setXxx(...)** (+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 récupérera 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 récupérera 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" (nouvelles petites classes avec attributs "nom" et "prefecture" "..." et méthode toString() ). Ces petits objets pourront également correspondre à la partie interne des noeuds de l'arbre (DefaultMutableTreeNode 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.