

Cours Programmation Java

v 1.4

Mohammed Lahmer (©2002-2023)
M.lahmer@umi.ac.ma



- Méthodes static et default dans une Interface (p.24)
- Intégration du Lambda Language (p.29 et p. 47)
- Ajout d'un chapitre sur le JavaFX (Chapitre 7)

SOMMAIRE

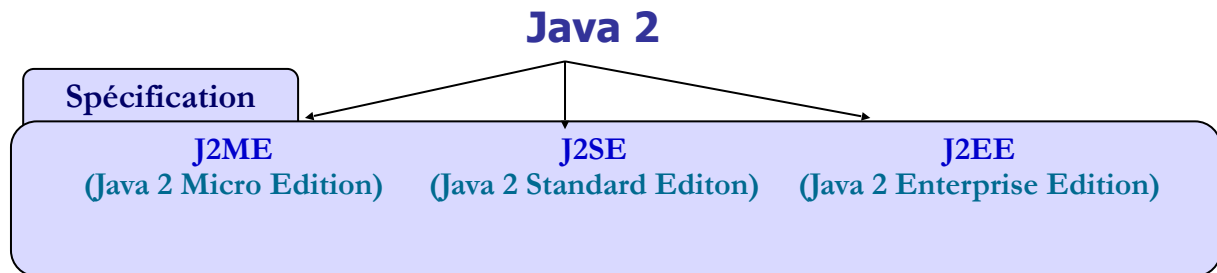
SOMMAIRE.....	2
1. Présentation du Langage	4
1. Bref historique	4
2. Les caractéristiques du langage Java	4
3. Kit de développement	6
2. Structure du Langage	9
1. La syntaxe	9
2. Classes et Objets	12
2. Méthodes et variables de la classe	14
3. Les packages	15
4. Classes particulières	16
3. L'héritage en Java	21
1. Principe de L'héritage	21
2. Syntaxe de l'héritage	21
3. Polymorphisme	22
4. Classes et méthodes abstraites	23
5. Les interfaces	24
4. Les Collections.....	25
1. Définitions	25
2. L'interface Collection	26
3. La classe ArrayList	27
4. L'interface Map.....	28
5. Tri et Recherche dans des collections	29
6. La gestion du temps (Date/Time)	30
4. Les entrées sorties.....	31
1. Les flux java	31
2. La classe File	31
3. Les flux octets.....	32
4. Les flux caractères	34
5. Les filtres	35
6. Sérialisation des objets	35
6. Le graphique en Java.....	37
1. Graphique en Java.....	37
2. Composants lourds/ légers.....	39
3. Composants de base.....	40
4. Mise en page des composants dans un conteneur.....	42
5. Traitement des événements.....	44
6. Les autres Composants graphiques.....	47
7. JavaFX.....	55
1. Introduction	55
2. Architecture	55
3. Application JavaFX.....	56
4. Exemple	59
5. ListView	60

6. Text Style	61
7. Charts	61
8 Accès aux bases de données en JDBC	63
1. Principe	63
2. Mise en œuvre	63
3. PreparedStatement et CollablStatement	65
4. Les métadonnées	66
9. Client / Serveur en RMI.....	67
1. Définition	67
2. Mise en œuvre	67

1. Présentation du Langage

1. Bref historique

- **1993** : projet Oak (langage pour l'électronique grand public)
- **1995** : Java / HotJava
- **Mai 95** : Netscape prend la licence
- **Sept. 95** : JDK 1.0 b1
- **Déc. 95** : Microsoft se dit intéresser
- **Janv. 96** : JDK 1.0.1
- **Été 96** : Java Study Group ISO/IEC JTC 1/SC22
- **Fin 96** : RMI, JDBC, JavaBeans, ...
- **Fév. 97** : JDK 1.1
- **98** : C'est la grande révolution JDK 1.2 ou java 2



- **J2SE 1.3** **2000**
- **J2SE 1.4** **2002** (instruction assert, intégration XML et des extensions de sécurité)
- **J2SE 5.0** **2004** (annotation@, type enum, varargs, for each)
- **Java SE 6** **2006** (support du web services, amélioration du graphique)
- **Java SE 7** **2011** (String dans switch, nouveau flux I/O)
- **Java SE 8** **2014** (JavaFX nouveau mode de développement du graphique, fonction Lambda, nouvelle api Date/Time)

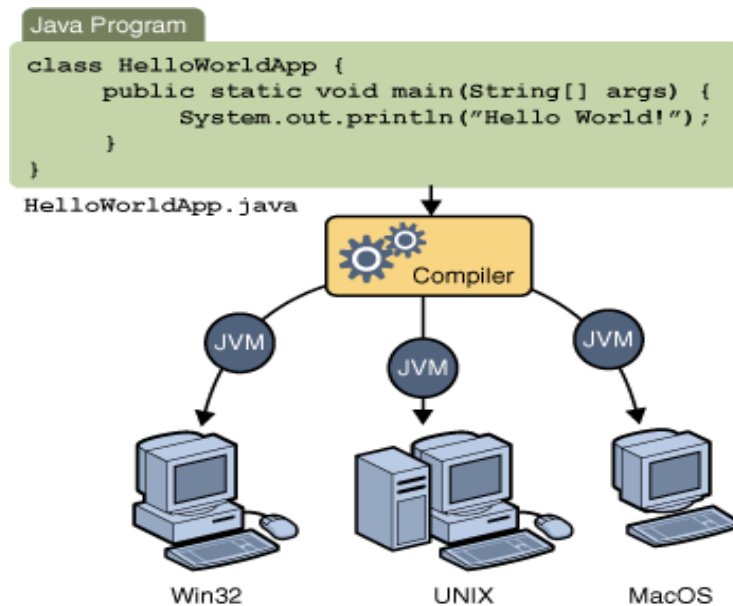
2. Les caractéristiques du langage Java

1. Java est un langage orienté objets

- Tout est classe (pas de fonctions à l'extérieure d'une classe) sauf les types primitifs (int, float, double, ...) et les tableaux
- Toutes les classes dérivent de java.lang.Object
- Héritage simple pour les classes
- Héritage multiple pour les interfaces
- Les objets se manipulent via des références
- Une API objet standard est fournie
- La syntaxe est proche de celle de C

2. Langage semi-compilé et semi-interprété

- Un langage compilé est un langage qui permet de créer un fichier exécutable à partir d'un programme source
- Un langage interprété est un langage où le code source doit être lu par un interpréteur pour s'exécuter.
- Le code source est d'abord compilé en byte code (attention ce n'est pas un code machine) puis interprété et exécuté par la machine virtuelle Java (JVM)



3. Java est portable

- Les programmes java (à part certains cas très particuliers) sont indépendants de la plateforme.
- Il est d'abord traduit dans un langage appelé « **bytecode** » qui est le langage d'une machine virtuelle (JVM ; *Java Virtual Machine*) dont les spécifications ont été définies par *Sun*.
- La *Java Virtual Machine* (JVM) est présente sur Unix, Linux, Win32, Mac, OS/2, Mozilla, Netscape, IE, ...
- La taille des types primitifs est indépendante de la plateforme contrairement à C ou C++.
- Java supporte un code source écrit en Unicode facilement internationalisable.

4. Java est robuste

- Gestion de la mémoire par un *garbage collector* qui libère la mémoire d'un objet non utilisé. Évite la surcharge
- Mécanisme d'exception des erreurs système pris en charge par la JVM. (Accès à une référence null exception)

- Compilateur contraignant (erreur si exception non gérée, si utilisation d'une variable non affectée, ...).
- Tableaux = objets (taille connue, débordement : exception)
- Contrôle des *cast* à l'exécution. Seules les conversions sûres sont automatiques.

5. Java est sécurisé

La sécurité est indispensable avec le code mobile (Applications répartis). Il existe trois niveaux de sécurité au niveau de la JVM:

- *Vérification* : vérifie le *byte code*.
- *Chargement (Class Loader)* : responsable du chargement des classes. (vérifier les autorisations *private*, *protected*, *public*)
- *Contrôle (Security Manager)* : contrôle l'accès aux ressources. En plus vous avez la possibilité de certifier le code par une clé. (mécanismes de cryptographie et d'authenticité)

Une fois le code est chargé en mémoire, il est impossible de faire un accès direct à la mémoire contrairement à C et C++.

6. Java est multi-thread et Distribué

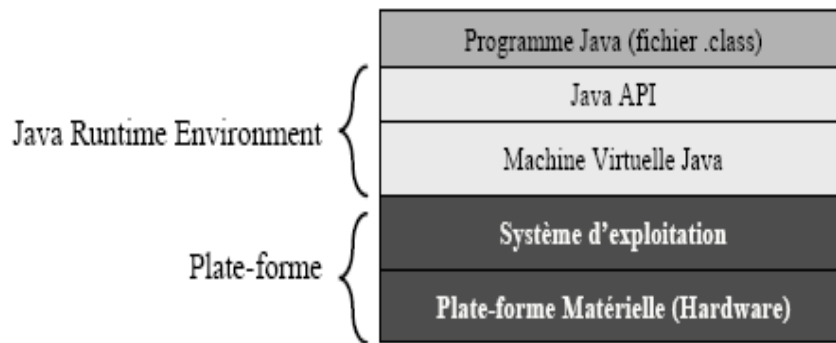
- Java est multi-thread
 - Accès concurrents à objet gérés par un *monitor* (*verrou*, *sémaphore*, ..).
 - Implémentation propre a chaque JVM->Difficultés pour la mise au point et le portage.
- Java est distribué
 - API réseau (*java.net.Socket*, *java.net.URL*, ...).
 - Chargement / génération de code dynamique.
 - Applet.
 - Servlet.
 - (RMI) Remote Method Invocation.
 - JavaIDL (CORBA).

3. Kit de développement

1. JVM / JRE

Les machines virtuelles java sont fournies gratuitement par Sun Microsystems. Elles sont fournies sous forme de JRE (Java Runtime Environment) comprenant la machine virtuelle et les package du noyau de java (APIs).

JRE = JVM + APIs Noyau



Le JRE ne contient pas d'outils ou d'utilitaires pour compiler ou déboguer les applets ou les applications.

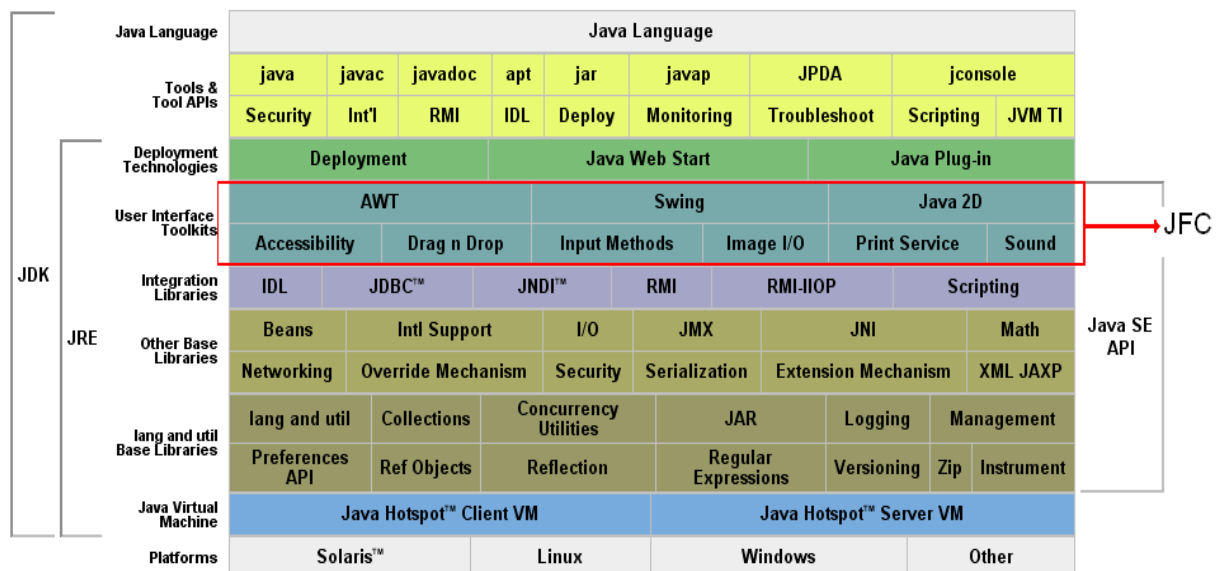
Intel, Motorola, Ultra-sparc sont des exemples de processeurs intégrés dans des plate-formes type : Pc, Mac ou Station Sun

2. Structure JDK

JSDK (Java Software Development Kit) est composé de JRE et plusieurs outils.

JSE = JRE + Outils

Java™ SE 6 Platform at a Glance



Les outils

- **javac** : compilateur de sources java
- **java** : interpréteur de *byte code*
- **appletviewer** : interpréteur d'applet
- **javadoc** : générateur de documentation (HTML, MIF)
- **javah** : générateur de *header* pour l'appel de méthodes natives

- **javap** : désassembleur de *byte code*
- **jdb** : débogueur
- **javakey** : générateur de clés pour la signature de code
- **rmic** : compilateur de *stubs* RMI
- **rmiregistry** : "*Object Request Broker*" RMI

2. Les APIs de base

- **java.lang** : Types de bases, Threads, ClassLoader, Exception, Math, ...
- **java.util** : Hashtable, Vector, Stack, Date, ...
- **java.applet**
- **java.awt** : Interface graphique portable
- **java.io** : accès aux i/o par flux, wrapping, filtrage
- **java.net** : Socket (UDP, TCP, multicast), URL, ...
- **java.lang.reflect** : introspection sur les classes et les objets
- **java.beans** : composants logiciels réutilisables
- **java.sql** (JDBC) : accès homogène aux bases de données
- **java.security** : signature, cryptographie, authentification
- **java.serialisation** : sérialisation d'objets
- **java.rmi** : *Remote Method Invocation*
- **Javax.swing** : extension du java.awt, plus riches en terme de composants
- **Java2D** : intégré depuis JDK 1.2 sous forme de sous paquetage du awt.
- **Java.security** et **Javax.security** : contient une implémentation de la plupart des standard de la cryptographie et le hashage.
- **javax.xml** : pour le traitement des fichiers xml

Il existe d'autres Apis tierces que vous pouvez télécharger indépendamment du Kit :

- **JFreeChart / JGraph** : pour la génération des graphes tels que les histogrammes, les secteurs,....
- **SNMP4J** : La supervision d'un réseau par le biais du protocole snmp
- **Jpcap** : La capture des trames, paquets qui circules dans un réseau
- **org.apache.commons.net** : offre un support à la majorité des protocoles réseaux (FTP/FTPS, NNTP, SMTP(S), POP3(S), IMAP(S), Telnet, TFTP, ...)

2. Structure du Langage

1. La syntaxe

1. Premier programme Java

On crée un fichier `HelloWorld.java` qu'on met directement dans `c` :

```
public class HelloWorld {  
public static void main(String[] args)  
{  
    System.out.println("Hello world"); // Afficher sur Ecran  
}}
```

- La classe `HelloWorld` est public, donc le fichier qui la contient doit s'appeler (en tenant compte des majuscules et minuscules) `HelloWorld.java`
- Un fichier `.java` peut contenir la définition de plusieurs classes mais une et une seule qui doit être publique
- **Compilation avec javac → `c:\>javac HelloWorld.java`**
 - crée un fichier « `HelloWorld.class` » qui contient le *bytecode*, situé dans le même répertoire que le fichier « `.java` »
- **Execution avec java → `c:\>java HelloWorld`**
 - interprète le *bytecode* de la méthode `main()` de la classe `HelloWorld`
- `HelloWorld.class` doit être dans le répertoire courant ou dans un des emplacements indiqués par une option `-classpath` ou par la variable `CLASSPATH`

2. Les identificateurs et les commentaires

- Un identificateur Java
 - est de longueur quelconque
 - commence par une lettre Unicode (caractères ASCII recommandés)
 - peut ensuite contenir des lettres ou des chiffres ou le caractère souligner `_`
 - ne doit pas être un mot-clé ou les constantes `true`, `false` et `null`
- Les commentaires
 - Sur une seule ligne `int prime = 1500; // prime fin de mois`
 - Sur plusieurs lignes : `/* Première ligne du commentaire`
`Suite du commentaire */`

3. Les types primitifs

- `boolean`(`true/false`), `byte` (1 octet), `char` (2 octets), `short` (2 octets), `int` (4 octets), `long` (8 octets), `float` (4 octets), `double` (8 octets).
- Les caractères sont codés par le codage Unicode (et pas ASCII) `'\u03a9'`

- Les variables peuvent être déclarées n'importe où dans un bloc.
- Les affectations non implicites doivent être *castées* (sinon erreur à la compilation).

```
int i = 258;
long l = i;
```

- Valeurs par défaut

Si elles ne sont pas initialisées, les variables d'instance ou de classe (pas les variables locales d'une méthode) reçoivent par défaut les valeurs suivantes :

boolean false

char '\u0000'

Entier (**byte short int long**) **0 0L**

Flottant (**float double**) **0.0F 0.0D**

Référence d'objet **null**

Les constantes

- Une constante « entière » est de type **long** si elle est suffixée par « L » et de type **int** sinon
- Une constante « flottante » est de type **float** si elle est suffixée par « F » et de type **double** sinon : **.123587E-25F** // de type float

4. La conversion de type

- En Java, **2 seuls cas** sont autorisés pour les *casts* :
 - entre types primitifs (suit les mêmes règles que C),
 - entre classes mère/ancêtre et classes filles (voir plus loin dans ce cours)
- Un *cast* entre types primitifs peut occasionner une perte de données Par exemple, la conversion d'un **int** vers un **short**
- Un *cast* peut provoquer une simple perte de précision. Par exemple, la conversion d'un **float** vers un **int**

```
int i = 258;
long l = i; // ok
byte b = i; // error: Explicit cast needed to convert int to byte
byte b = 258; // error: Explicit cast needed to convert int to byte
byte b = (byte)i; // ok mais b = 2
```

5. Les structures de contrôle

- Essentiellement les mêmes qu'en C
 - if, switch, for, while, do while
 - ++, +=, &&, &, <<, ?:, !, ~, ^ ...
- Plus les blocs labellisés

```
UN: while(...) {
DEUX: for(...) {
```

```
TROIS: while(...) {
if (...) continue UN;// Reprend sur la première boucle while
if (...) break DEUX;// Quitte la boucle for
continue;// Reprend sur la deuxième boucle while
}}
```

- Une boucle for spéciale a été rajoutée depuis la version 1.5 afin de faciliter le parcours des tableaux d'objet (voir exemple Tableau paragraphe 6.)

- A partir de JDK 1.8, il est désormais possible de faire un switch sur un type String

Switch (jour) {

case "Lundi" :

....

}

6. Les tableaux

En Java un tableau est traité comme un objet, donc il faut le déclarer et l'initialiser avant de l'utiliser

■ Déclaration

int[] tab_of_int; // équivalent à : int tab_of_int[];

Color rgb_cube[][][];

■ Création et initialisation

array_of_int = new int[42];

rgb_cube = new Color[256][256][256];

int[] primes = {1, 2, 3, 5, 7, 7+4};

tab_of_int[0] = 3 ;

■ Utilisation

int l = tab_of_int.length; // l = 42

L'indice commence à **0** et se termine à **length - 1**

int e = tab_of_int[50]; /* Lève une Exception **ArrayIndexOutOfBoundsException** */

Exemple de programme qui affiche les éléments d'un tableau

```
public class AfficherTab {
public static void main(String[] args)
{
int [ ] T={10, 2, 15, -8, 7};
for (int el:T)
{
System.out.println(el);
}
}
}
```

Il existe désormais plusieurs fonctions prédéfinies qui permettent par exemple de :

- copier une partie d'un tableau dans un autre
 - La fonction **arraycopy** de la classe **System**

- **public static void arraycopy(Object src, int src_position, Object dst, int dst_position, int length)**
- comparer 2 tableaux
 - La fonction **equals()** de la classe **java.util.Arrays**
 - **public static boolean equals(double[] a1, double[] a2)**

Les paramètres de ligne de commande ou les arguments de main constituent un tableau de chaînes de caractères

```
class Arguments {
public static void main(String[] args) {
for (int i=0; i < args.length; i++)
    System.out.print(args[i]);
}}
java Arguments toto titi //affichera toto puis titi
```

2. Classes et Objets

1. Objet / Classe

- Un objet peut être envisagé de façon pragmatique comme constitué de deux parties :
 - Une partie de stockage physique (Objet)
 - Une partie descriptive (Classe)
- L'Objet peut être apparenté à une variable de type structure en C (record en pascal, ligne d'une table dans une BD Relationnelle)
- Une classe n'est pas destinée au stockage des données. C'est un type de données abstrait, caractérisé par des propriétés (attributs et méthodes) communes à des objets et permettant de créer des objets possédant ces propriétés.

Nom_Classe
Attributs
Constructeurs
Méthodes

Point
float x float y
Point(int x,int y)
Void afficher(Point p)

Les objets sont construits à partir d'une classe en utilisant un constructeur.

2. Les constructeurs

Chaque classe a un ou plusieurs constructeurs qui servent à

- créer les instances
- initialiser l'état de ces instances

Un constructeur a le même nom que la classe et n'a pas de type retour. Lorsque le code d'une classe ne comporte pas de constructeur, un constructeur sera automatiquement ajouté par Java. Pour une classe **Nom_Classe**, ce constructeur par défaut sera **[public] Nom_Classe() {}**

Exemple

```
Fichier Point.java
/** Modélise un point de coordonnées x, y */
public class Point {
    private int x, y; // Les attributs
    // Le constructeur
    public Point(int x1, int y1) {
        x = x1; y = y1;
    }
    //Methode
    public double distance(Point p) {
        return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
    }
}

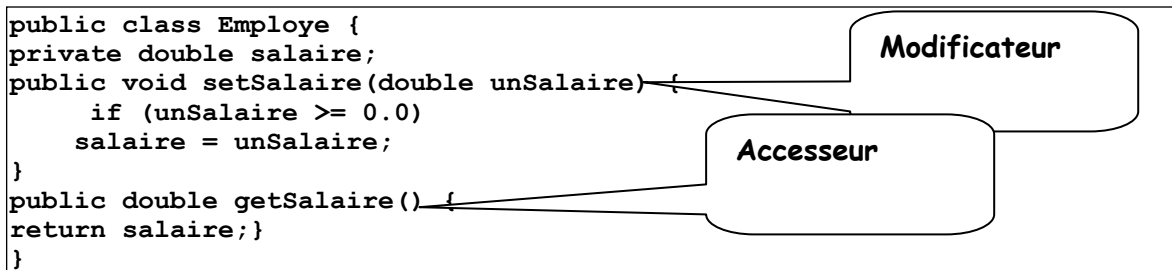
Fichier TestPoint.java
/** Pour tester la classe Point */
class TestPoint {
    public static void main(String[] args) {
        Point p1 = new Point(1, 2);
        Point p2 = new Point(5, 1);
        System.out.println("Distance : " + p1.distance(p2));
    }
}
```

Le mot clé new permet d'allouer de l'espace mémoire au nouvel objet de type Point.

3. Les méthodes

Une méthode est l'équivalent d'une fonction en C. Quand la méthode renvoie une valeur, on doit indiquer le type de la valeur renvoyée dans la déclaration de la méthode **double calculSalaire(int indice, double prime)**. Le pseudo-type **void** indique qu'aucune valeur n'est renvoyée **void setSalaire(double unSalaire)**

```
public class Employe {
    private double salaire;
    public void setSalaire(double unSalaire) {
        if (unSalaire >= 0.0)
            salaire = unSalaire;
    }
    public double getSalaire() {
        return salaire;
    }
}
```



4. Surcharge d'une méthode

Signature d'une méthode : nom de la méthode et ensemble des types de ses paramètres :

Signature de la méthode **main()** : **main(String[])**

En Java, on peut surcharger une méthode, c'est-à-dire, ajouter une méthode qui a le même nom mais pas la même signature qu'une méthode existante :

- **calculerSalaire(int)**
- **calculerSalaire(int, double)**

Il est interdit de surcharger une méthode en changeant le type de retour par exemple, il est interdit d'avoir ces 2 méthodes dans une classe :

- **int calculerSalaire(int)**
- **double calculerSalaire(int)**

5. Passage des paramètres

Le mode de passage des paramètres dans les méthodes dépend de la nature des paramètres :

- par référence pour les objets
- par copie pour les types primitifs

```
public class C {  
void methode1(int i, StringBuffer s)  
{ i++; s.append("d");}  
void methode2() {  
int i = 0;  
StringBuffer s = new StringBuffer("abc");  
methode1(i, s);  
System.out.println(i=" + i + ", s=" + s); // i=0, s=abcd    }  
}
```

Du moment que I est de type primitif il est passé à méthode1 par copie et donc sa valeur n'a pas changée. Mais s est un Objet donc il prend la dernière valeur attribuée dans methode1 parce qu'il est transféré par référence.

2. Méthodes et variables de la classe

1. Variables de classe

Certaines variables peuvent être partagées par toutes les instances d'une classe. Ce sont les variables de classe (modificateur static en Java). Si une variable de classe est initialisée dans sa déclaration, cette initialisation est exécutée une seule fois quand la classe est chargée en mémoire

Les variables static sont communes à toutes les instances de la classe.

```
public class Employe {  
private static int nbEmployes = 0;  
// Constructeur  
public Employe(String n, String p) {  
nom = n;prenom = p;nbEmployes++;}  
  
public static void main(String[] args){}  
Employe e1=new Employe("Toto","Titi");  
System.out.println("Nombre Employer est: "+ Employe.nbEmployer );}}
```

Les blocs static permettent d'initialiser les variables static trop complexes à initialiser dans leur déclaration. Ils sont exécutés une seule fois, quand la classe est chargée en mémoire

```
class UneClasse {  
private static int[] tab = new int[25];  
static {  
for (int i = 0; i < 25; i++) {  
tab[i] = -1;  
}  
....  
} // fin du bloc static
```

2. Méthodes de classe

Une méthode de classe (modificateur **static** en Java) exécute une action indépendante d'une instance particulière de la classe. Elle ne peut utiliser de référence à une instance courante (**this**)

- **static double tripleSalaire() {return salaire * 3;} // NON**

La méthode **main()** est nécessairement **static**. Pourquoi ?

Pour désigner une méthode **static** depuis une autre classe, on la préfixe par le nom de la classe :

- **int n = Employe.nbEmploye();**

On peut aussi la préfixer par une instance quelconque de la classe (à éviter car cela nuit à la lisibilité : on ne voit pas que la méthode est **static**) : **int n = e1.nbEmploye();**

3. Variable d'état final

Le modificateur **final** indique que la valeur de la variable (d'état ou locale) ne peut être modifiée. On pourra lui donner une valeur une seule fois dans le programme (à la déclaration ou ensuite). Une variable de classe **static final** est vraiment constante dans tout le programme ;

static final double PI = 3.14;

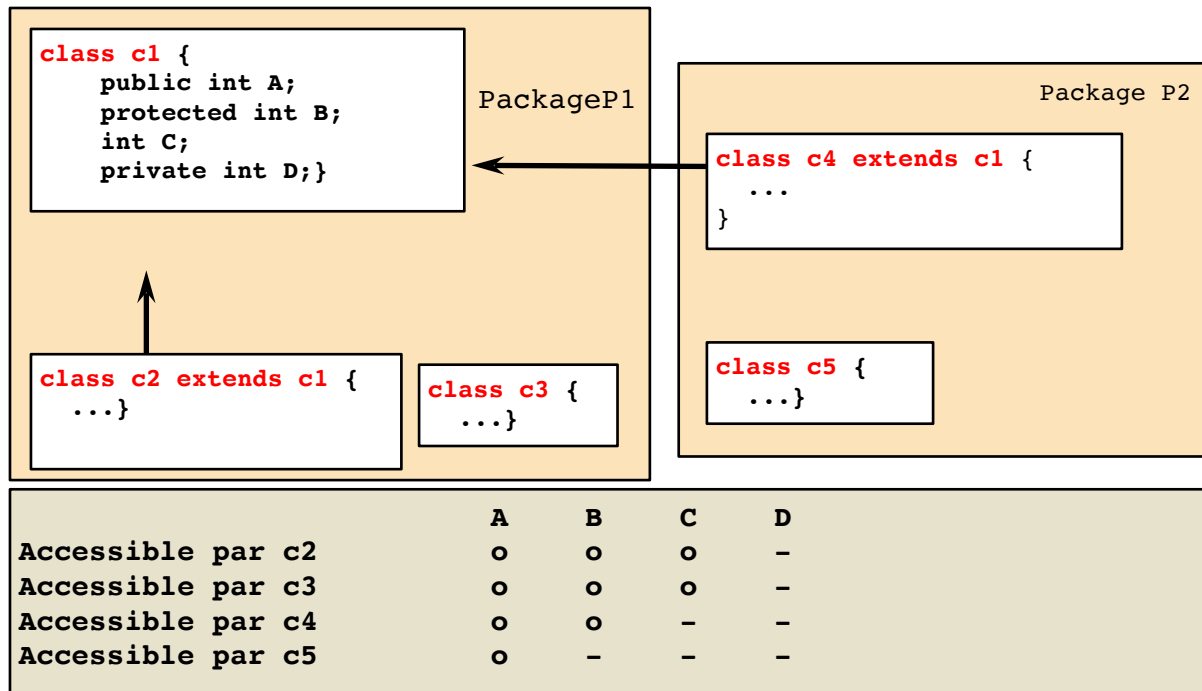
Une variable *d'instance* (pas **static**) **final** est constante pour chaque instance ; mais elle peut avoir 2 valeurs différentes pour 2 instances. Elle peut ne pas être initialisée à sa déclaration mais elle doit avoir une valeur à la sortie de tous les constructeurs

3. Les packages

Un package regroupe un ensemble de classes sous un même espace de 'nomage'. Les noms des packages suivent le schéma : name.subname ...

Une classe **employe** appartenant au package **societe.personnel** doit se trouver dans le fichier **societe\travail**. L'instruction **package** indique à quel package appartient la ou les classe(s) de l'unité de compilation (le fichier). En dehors du package, les noms des classes sont : **packageName.className**, sinon L'instruction **import packageName** permet d'utiliser des classes sans les préfixer par leur nom de package.

Encapsulation (public, protected et private)



4. Classes particulières

1. Les chaînes de caractères

Il existe deux classes pour la manipulation des chaînes de caractères :

- **String** pour les chaînes constantes
- **StringBuffer** pour les chaînes variables

On utilise le plus souvent **String**, sauf si la chaîne doit être fréquemment modifiée.

L'affectation d'une valeur littérale à un **String** (pas à un **StringBuffer**) s'effectue par :

chaîne = "Bonjour";

La spécification de Java impose que

- **chaîne1 = "Bonjour";**
- **chaîne2 = "Bonjour";**

crée un seul objet **String** (référéncé par les 2 variables)

La Concaténation

La concaténation se fait par l'opérateur + :

- **String s = "Bonjour" + " les amis";**
- **int x = 5;**
- **s = "Valeur de x + 1 = " + x + 1;**

Les types primitifs lorsqu'ils sont concaténés avec un message sont convertis en **String**, ainsi s contient **Valeur de x + 1 = 51**

Attention, la concaténation de **Strings** est une opération coûteuse (elle implique en particulier la création d'un **StringBuffer**). Il faut passer explicitement par un **StringBuffer** si la concaténation doit se renouveler. Une seule création d'un **StringBuffer** pour toutes les concaténations


```
char[] t={'a','b','r','c','e','f'};
c = new StringBuffer(t[0]);
for (int i = 1; i < t.length; i++) c.append(t[i]);
```

Fonctions de comparaison

La méthode **equals()** teste si 2 instances de **String** contiennent la même valeur elle retourne true ou false:

```
String s1, s2;
```

```
s1 = "Bonjour ";s2 = "les amis";
```

```
if ((s1 + s2).equals("Bonjour les amis"))
```

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

- **equalsIgnoreCase()** pas de différence entre majuscule et minuscule
- **s.compareTo(t)** renvoie (« signe » de **s-t**)
 - 0 en cas d'égalité de s et de t,
 - un nombre entier positif si s suit t dans l'ordre lexicographique
 - un nombre entier négatif sinon
- **compareToIgnoreCase(t)**

D'autres fonctions

Fonction	Rôle
int length()	Retourne la longueur de la chaîne appelante
String toString()	Convertir un objet en String
char charAt(int i)	Retourne le caractère à la position i (0<= i < length())
String substring (int d, int f)	Retourne tous les caractères entre la position d et f dans une chaîne.
int indexOf(String sCh)	Retourne la position de la sous chaîne sCh s'elle existe. -1 si non
int indexOf(String sCh, int debut)	Cherche la position de sCh à partir de debut. elle retourne sa position s'elle existe et - 1 si non
String [] split(String sep)	Permet de décomposer une chaîne en un tableau de String en utilisant le séparateur sep

▪ Exercice

Ecrire un programme Java qui permet de récupérer le protocole, la machine, le répertoire et le nom du fichier une adresse Web <http://www.iam.ma/rep1/rep2/fichier.html>

2. Les classes enveloppes

Le paquetage **java.lang** fournit des classes pour envelopper les types primitifs (les objets sont constants) : **Byte, Short, Integer, Long, Float, Double, Boolean, Character.**

les classes enveloppes offre des méthodes utilitaires (le plus souvent **static**) pour faire des conversions entre types primitifs (et avec la classe **String**)

Fonction	Classe	Rôle
int intValue()	Integer	Retourne l'entier encapsulé dans l'objet de type Integer
static int parseInt(String ch)		Convertis une chaîne de caractère ch en un entier
static String toBinaryString(int i)	Integer	Convertis un entier en String représentant la représentation binaire de i.
static String toHexString(int i)	Integer	Convertir un entier en String représentant la représentation Hexa Décimal de i.
static String valueOf(int i)	String	Convertis un entier en String.
static String toString(char c)	Character	Convertis un caractère c en String

3. La classe System

Cette classe est très utile pour diverses opérations d'entrée et sortie, d'émission d'erreurs, d'accès aux propriétés du système, et bien d'autres fonctionnalités relatives à des tâches système.

- **System.getProperty(NomPropriete)** : renvoie la valeur de la propriété system ci-dessous quelques exemples de Récupération des variables d'environnement :
 - Nom Utilisateur Connecté : `System.getProperty("user.name")`
 - Répertoire de travail : `System.getProperty("user.home")`
 - Nom OS installé : `System.getProperty("os.name")`
 - Version OS installé : `System.getProperty("os.version");`
- **System.exit(code retour)** : arrêt immédiat de la JVM
- **System.gc()** : appel explicite du Garbage Collector ou (**Ramasse Miettes**). Le ramasse-miettes (garbage collector) est une tâche(job) qui travaille en arrière-plan et dont le rôle de libérer la place occupée par les instances non référencées. Il intervient automatiquement quand le système a besoin de mémoire ou, de temps en temps, avec une priorité faible (voir Thread). Avant la suppression d'un objet par le ramasse miettes, le système appelle la méthode **finalize()** de la classe de l'objet (si elle existe).

```
class garbage{
static boolean gcok=false; /*devient true lorsque le GC est
                           appelé*/
static int count;
public garbage(){count++;}
public void finalize(){
```

```
if(gcok==false)
System.out.println("Appel GC:\n"+"Nombre d'instance
crees/supprimées : "+count);
gcok=true;}
public static void main(String[] args)
while(!gcok) {new garbage();}}
```

4. La classe exception et ces dérivées

Les exceptions permettent de traiter séparément les erreurs pouvant survenir dans un bloc d'instructions. Ce sont des instances de classes dérivant de **java.lang.Exception**. L'appel à une méthode pouvant lever une exception doit :

- soit être contenu dans un bloc **try/catch**

```
try {
/* Code pouvant lever des IOException ou des SecurityException*/
}
catch (IOException e) {
/* Gestion des IOException et des sous-classes de
IOException*/}
catch (Exception e){
// Gestion de toutes les autres exceptions}
finally{
// Code qui sera exécuté dans tous les cas}
```

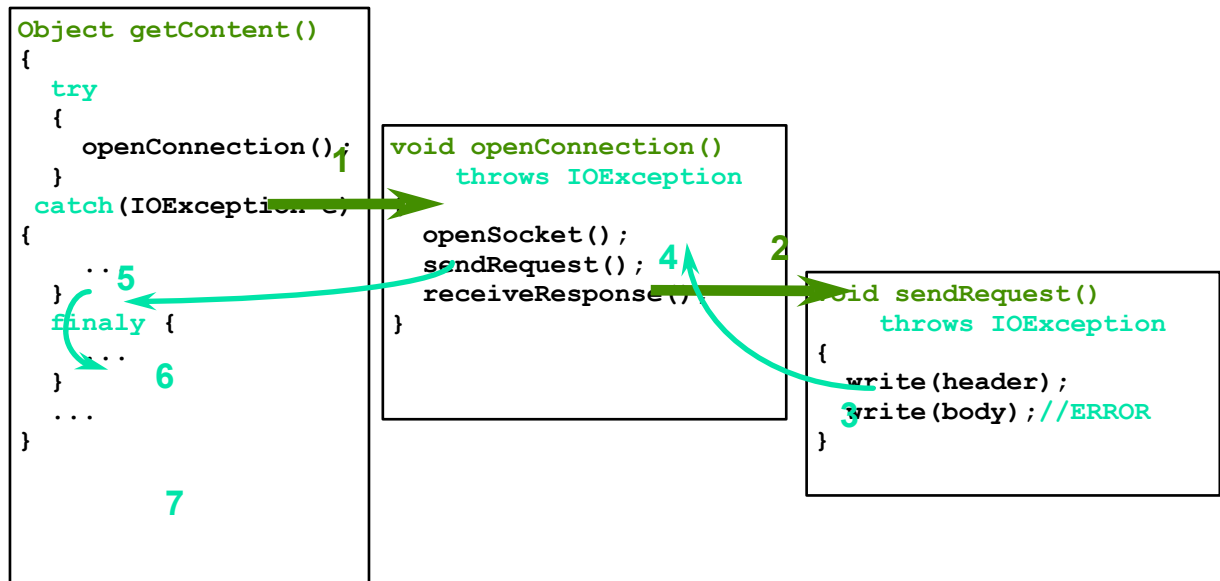
Noter bien que finally est optionnel. Il est utilisé lorsque vous voulez exécuter un bloc d'instruction indépendamment du résultat de try/catch.

- soit être situé dans une méthode propageant (**throws**) de cette classe d'exception **void fonction throws IOException { // ... }**

Plusieurs exceptions sont déjà définies dans java.lang comme par exemples :

- NullPointerException
- OutOfMemoryException
- ArithmeticException
- ClassCastException
- ArrayIndexOutOfBoundsException

La levée d'une exception provoque une remontée dans l'appel des méthodes jusqu'à ce qu'un bloc catch acceptant cette exception soit trouvé. Si aucun bloc catch n'est trouvé, l'exception est capturée par l'interpréteur et le programme s'arrête.



Il est tout à fait possible de définir vos propres exceptions en créant une classe qui dérive de la classe mère Exception.

```
public class MonException extends Exception
{
    public MonException(String text)
    {
        super(text);
    }
}
```

On utilise le mot clé **throw** pour lever une exception en instanciant un objet sur une classe dérivée de la classe Exception : **throw new** MonException("blabla");

3. L'héritage en Java

1. Principe de L'héritage

Le concept d'héritage est un des concepts les plus importants de l'orienté objet. C'est grâce à ce dernier que la programmation objet est ce qui est aujourd'hui. Par définition l'héritage est le fait de pouvoir utiliser tout ce qui a déjà été fait ([réutilisation](#)) et de pouvoir l'enrichir ([spécialisation](#)). Il indique toujours une relation de type "est un" ou « est une sorte de » :

- Un lapin est un animal,
- Un employé est une personne,
- Un pc est une machine.

En Java, chaque classe a une et une seule classe mère (pas d'héritage multiple) dont elle hérite les variables et les méthodes. Le mot clef **extends** indique la classe mère par exemple : **class B extends A{ }**. La classe A s'appelle une classe mère, ou super-classe et la classe B qui hérite de la classe A s'appelle une classe fille ou sous-classe.

Dans B on indique seulement ce qui a changé dans B par rapport au code de A ; on peut :

- **ajouter** des variables, des méthodes et des constructeurs
- **redéfinir** des méthodes (exactement la même signature et le même type retour)
- **surcharger** des méthodes (même nom mais pas même signature) (possible aussi à l'intérieur d'une classe)

Note : Si « B extends A », le grand principe est que tout B est un A

2. Syntaxe de l'héritage

1. Constructeur

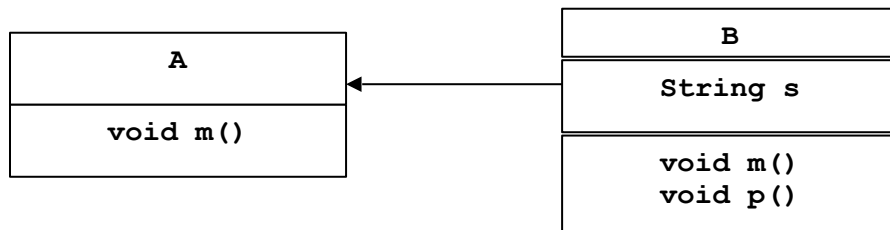
La règle générale est que pour construire un objet de la sous-classe il faut d'abord construire un objet de la super-classe. Si un constructeur de la classe dérivée appelle explicitement un constructeur de la classe de base, cet appel doit être obligatoirement la première instruction de constructeur. Il doit utiliser pour cela, le mot clé [super](#).

2. Règles de visibilité dans l'héritage

Si une classe **B** hérite de **A** (**B extends A**), elle hérite automatiquement et implicitement de tous les membres de la classe **A** (mais pas les constructeurs et les membres private)

3. Redéfinition

Supposons que **B** hérite de **A** et que la méthode **m()** de **A** soit redéfinie dans **B** selon le schéma UML suivant :



Un objet d'une classe dérivée peut toujours être utilisé au lieu d'un objet de sa super-classe donc on peut écrire par exemple :

- **A a = new B();**
- **a.m();**

Quelle méthode **m()** sera exécutée celle de **A** ou celle de **B** ? Éventuellement celle de **B** parce que le contenu de **a** est un objet de **B**.

4. Conversion

Un objet de type **B** est un **A**. Mais l'inverse n'est pas vrai. Une machine n'est pas nécessairement un pc.

A a ; B b = new B("s");

- **Cas a=b; // vrai**

L'écriture **a.m()** est vrai parce que **m()** est définie dans **A**. Mais **a** ne peut pas appeler **p()** donc **a.p()** est fausse.

- **Cas b=a; // fausse**

Dans ce cas un cast est nécessaire parce que **a** ne contient pas tous les éléments de **b** (dans ce cas le champs **s**)

- **Cas b=(B) a; // vrai lors de la compilation mais pas à l'exécution**

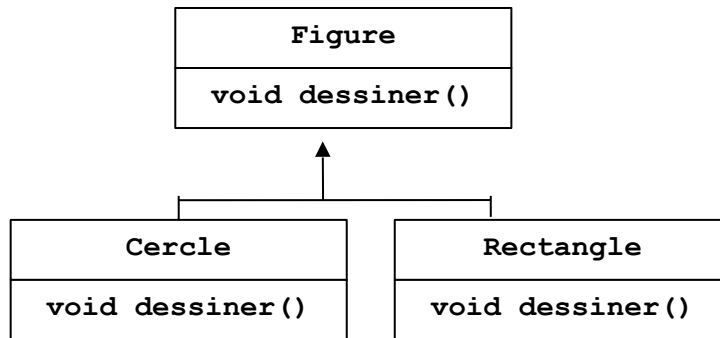
5. Le mot clé final

Une classe utilise le mot clef **final** dans sa définition ne peut avoir de classes filles (**String** et **java.util.Vector** sont **final**)

- Une Méthode **final** : ne peut être redéfinie
- Un Paramètre **final** (d'une méthode) : la valeur (éventuellement une référence) ne pourra être modifiée dans le code de la méthode
- Une Variable (locale ou d'état) **final** : la valeur ne pourra être modifiée après son initialisation

3. Polymorphisme

Une méthode est dite polymorphe si elle a le même nom dans plusieurs classes dérivées mais pas la même implémentation.



La façon de dessiner un cercle est différente de celle qu'on utilise pour dessiner un rectangle donc la méthode **dessiner()** est polymorphe.

<pre> public class Figure { public void dessiner() { } } public class Rectangle extends Figure { public void dessiner() { . . . } } public class Cercle extends Figure { public void dessiner() { ... } } </pre>	<pre> public class Dessin { private Figure[] figures; public void afficher() { for (int i=0; i < nbFigures; i++) figures[i].dessiner(); } public static void main(String[] args) { Dessin dessin = new Dessin(30); Cercle C ; Rectangle R ; ... dessin.ajoute(C) dessin.ajoute(R); dessin.afficher(); ...}} </pre>
--	---

Bien utilisé le polymorphisme permet de :

- éviter les codes qui comportent de nombreux branchements et tests. En effet sans polymorphisme le code de la méthode afficher() aurait du être comme suit :

```

for (int i=0; i < nbFigures; i++)
{if(figures[i] instanceof Cercle) dessiner_Cercle();
  if(figures[i] instanceof Rectangle) dessiner_Rectangle();
...}

```

Il faut savoir que les opérations comportant des ifs sont coûteuses au niveau processeur.

- faciliter l'extension des programmes : on peut créer de nouvelles sous classes sans toucher aux programmes déjà écrits. Par exemple, si on ajoute une classe Losange, le code de **afficher()** sera toujours valable.

4. Classes et méthodes abstraites

Une méthode est dite abstraite (modificateur **abstract**) si on la déclare, sans donner son implémentation : `public void payer()` ;

- Une classe doit être déclarée abstraite (**abstract class**) si elle contient au moins une méthode abstraite. Mais il est possible de déclarer une classe abstraite sans aucune méthode abstraite
- Il est interdit de créer une instance d'une classe abstraite.

Une méthode **static** ne peut être abstraite (car on ne peut redéfinir une méthode **static**)

<pre>abstract class Geometrie{ //Methode abstraite abstract double perimetre();} }</pre>	<pre>class Rectangle extends geometrie { double L,l; double perimetre(){ return L*l;}}</pre>
--	--

5. Les interfaces

Une interface est une « classe » purement abstraite dont toutes les méthodes sont abstraites et publiques (les mots-clés **abstract** et **public** sont optionnels dans la déclaration des méthodes). En fait, une interface correspond à un **service rendu** ou un comportement qui correspond aux méthodes de l'interface

[public] interface NomDeLInterface [extends InterfaceMere] {}

Une interface ne peut contenir que :

- des méthodes abstract et public
- des définitions de constantes publiques (« public static final »)

Une interface ne peut contenir de méthodes **static**, **final**, **synchronized**, **native** ou **default** (voire à la fin du chapitre)

Pour implémenter une interface, il faut rajouter à la déclaration de la classe concernée le mot clé **implements** suivi du nom de l'interface. Une classe peut implémenter une ou plusieurs interfaces (et hériter d'une classe)

<pre>public interface Figure { void dessineToi(); void deplaceToi(int x, int y); Position getPosition(); }</pre>	<pre>class rectangle implements Figure{ void dessineToi(){ /*Dessiner Rectangle*/} void deplaceToi(int x, int y){ /*Deplacer Rectangle*/} Position getPosition(){ /*Retourner la position du rectangle*/} }</pre>
--	---

Lorsqu'une classe C implémente une interface I elle doit redéfinir toutes les méthodes de I, sinon il faut la déclarer abstraite.

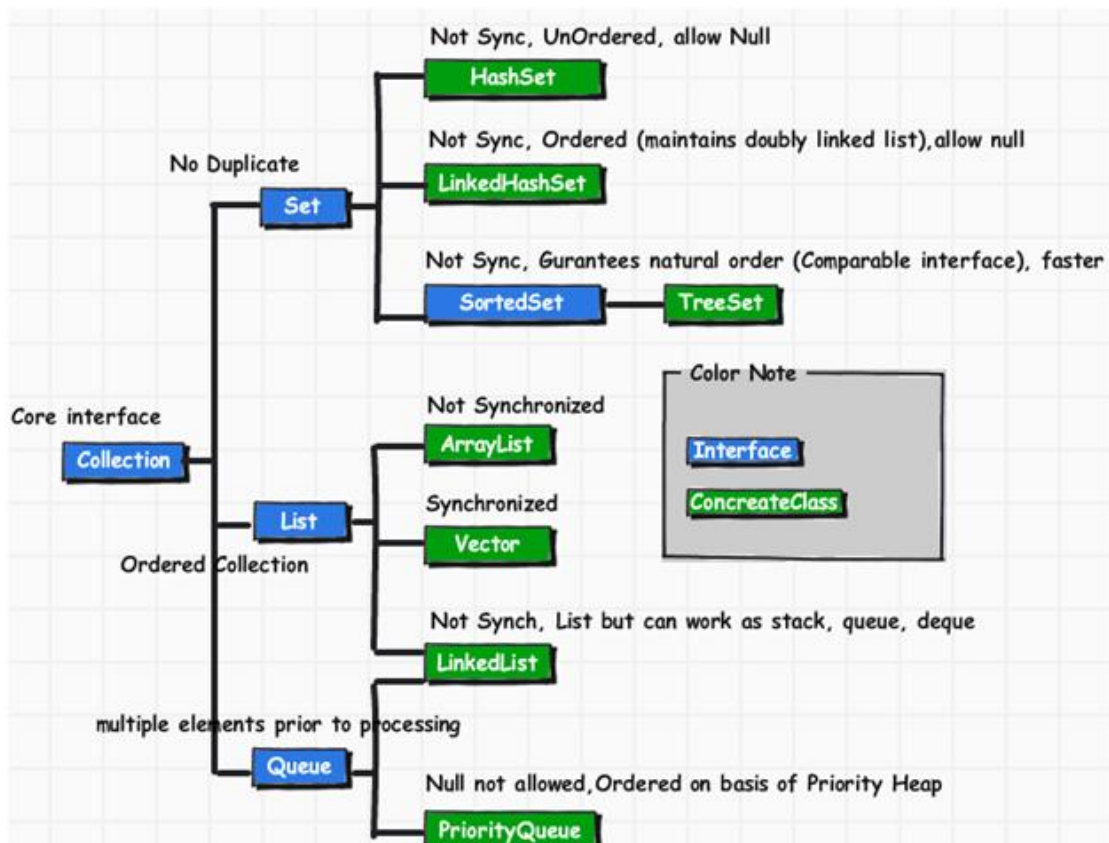
A partir de Java 8, il est désormais possible de définir dans une interface des méthodes de type default et ou static. Une méthode de type default peut être redéfinie mais une méthode static non.

4. Les Collections

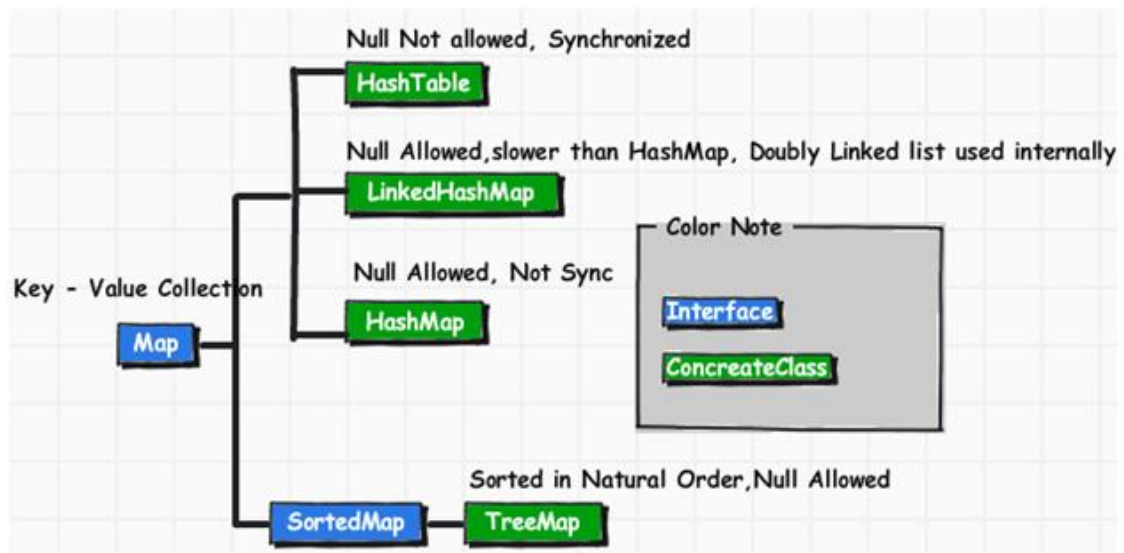
1. Définitions

Une collection est un objet qui contient d'autres objets par exemple, un tableau est une collection. Le JDK fournit d'autres types de collections sous la forme de classes et d'interfaces définies dans le paquetage **java.util** et organisées selon 2 hiérarchies d'interfaces principales

- **Collection** correspond aux interfaces des collections proprement dites



- **Map** correspond aux collections indexées par des clés ; un élément d'une *map* est retrouvé rapidement si on connaît sa clé.



2. L'interface Collection

1. Les méthodes d'une collection

boolean add(Object)	Ajoute un objet dans une collection
void clear()	Supprime tous les éléments du conteneur
boolean contains(Object)	Vérifie si l'objet argument est contenu dans la collection
boolean isEmpty()	Vrai si le conteneur ne contient pas d'éléments.
boolean remove(Object)	Supprime un objet de la collection
int size()	Renvoie le nombre d'éléments dans le conteneur.
Object[] toArray()	Renvoie un tableau contenant tous les éléments du conteneur.
Iterator iterator()	Renvoie un Iterator qu'on peut utiliser pour parcourir les éléments du conteneur.

2. Les itérateurs

Un itérateur (instance d'une classe qui implémente l'interface **Iterator**) permet d'énumérer les éléments contenus dans une collection. L'interface **Collection** fournit la méthode **Iterator iterator()** qui renvoie un itérateur pour parcourir les éléments de la collection. Les principales méthodes de **Iterator** sont :

- **boolean hasNext()** : permet de vérifier s'il y a un élément qui suit
- **Object next()** : retourne l'élément en cours et pointe sur l'élément suivant

```

Iterator it=maCollection.iterator();
while (it.hasNext()) // tant qu' il y a un element non parcouru
{
    Object o = it.next();
    //les opérations
}

```

3. La classe ArrayList

Est une classe qui implémente l'interface List. Une instance de la classe **ArrayList** est une sorte de tableau qui peut contenir un nombre quelconque d'instances de la classe **Object**. Les emplacements sont repérés par des nombres entiers ou des indices (à partir de 0). Les principales méthodes en plus de celle de Collection sont

- **Object get(int indice)**
- **int indexOf(Object obj)**
- **void add(int indice, Object element)**
- **void set(int indice, Object obj)**
- **Object remove(int index)**

Le type retour de la méthode **get()** est **Object** ; on doit donc souvent *caster* les éléments que l'on récupère dans un **ArrayList** si on veut les utiliser. Ce problème ne se pose plus dans JDK 1.5 et supérieur si on adopte le principe de généricité ou de template :

1) sans template

```

ArrayList l = new ArrayList(); // création d'une Liste l
l.add("Ahmed"); // ajouter les objets ds l
l.add("Mohamed");
l.add("Mustapha");
l.add("Taib");
Collections.sort(l); // trier les objets de l
String Nom =(String)l.get(2) ; // on utilise le cast

```

2) avec template

```

ArrayList<String>l=new ArrayList<String>(); // création d'une Liste l
// ajouter les objets ds l
Collections.sort(l); // trier les objets de l
String Nom =l.get(2) ; // Pas de cast

```

2. Parcours d'une ArrayList

En plus de l'Iterator, il est possible de parcourir une ArrayList en utilisant une simple boucle for ou en passant par l'interface ListIterator.

1) Boucle for

```

ArrayList L = new ArrayList(); // création d'une Liste L
L.add("Ahmed"); // ajouter les objets ds L
L.add("Mohamed");
L.add("Mustapha");
L.add("Taib");
Collections.sort(L);
for(String nom :L) system.out.println(nom);

```

2) ListIterator

ListIterator est une interface qui hérite de Iterator, ainsi en plus des méthodes hasNext() et next() elle dispose de deux autres méthodes supplémentaires hasPrevious() et previous() qui permettent de parcourir la liste dans le sens inverse.

```
List list = new ArrayList();
// ajouter les objets ds List
ListIterator iterator = list.listIterator(list.size());
while (iterator.hasPrevious()) {
    Object element = iterator.previous();
    // traitement d'un élément
}
```

4. L'interface Map

Une Map est un ensemble de couples objets (clé, valeur) où Deux clés ne peuvent être égales au sens de equals(). Dans l'algorithmique classique c'est une table de hachage (hashTable).

Clé (code Module)	Valeur(Module)
IF11	M11
IF12	M12

Une entrée de la table est représentée par l'interface Entry qui permet de manipuler les éléments d'une paire comme suit :

```
public interface Entry {
    Object getKey();
    Object getValue();
    Object setValue(Object value);
}
```

getKey() et getValue() retournent respectivement la clé et la valeur associée à l'entrée en cours.

Il existe deux implémentations de l'interface Map à savoir :

- **HashMap**, table de hachage ; garantit un accès en temps constant
- **TreeMap**, arbre ordonné suivant les valeurs des clés avec accès en log(n)

La comparaison des objets d'une Map utilise l'ordre naturel (interface **Comparable**) ou une instance de **Comparator**. Les principales méthodes d'une hashMap sont :

- **Object put(Object key, Object value)**
- **Object get(Object key)**
- **Object remove(Object key)**

2. Parcours d'une Map

On peut parcourir une Map en utilisant :

Les valeurs : La méthode **values()** permet de convertir une hashMap en une Collection ne contenant que les valeurs.

Les clés : La méthode **setKey()** permet de convertir une hashMap en un Set ensemble de clés.

```
Map hm = new HashMap();
Employe e = new Employe("Halim");
e.setMatricule("E125");
hm.put(e.matricule, e);
/*crée et ajoute les autres employés dans la table de hachage*/
```

```

Employee e2 = (Employee)hm.get("E369");
Collection elements = hm.values();
Iterator it = elements.iterator();
//Afficher tous les employes
while (it.hasNext()) {
    System.out.println(
        ((Employee)it.next()).getNom());}

```

5. Tri et Recherche dans des collections

La classe **Collections** contient des méthodes **static**, utilitaires pour travailler avec des collections :

- tris (sur listes)
- recherches (sur listes)
- copies
- minimum et maximum

a) Tri

Si **l** est une liste, on peut trier **l** par : **Collections.sort(l)**. La méthode **sort()** ne fonctionnera que si tous les éléments de la liste sont d'une classe qui implémente l'interface **java.lang.Comparable**. Toutes les classes du JDK qui enveloppent les types primitifs (**Integer** par exemple) implémentent l'interface **Comparable**. Il en est de même pour les classes du JDK **String**, **Date**, **BigInteger**, **BigDecimal**. Cependant si on veut trier une liste d'objets qui ne sont pas comparables par défaut il faut passer par l'interface **Comparator**.

Interface Comparator

Si les éléments de la collection n'implémentent pas l'interface **Comparable**, ou si on ne veut pas les trier suivant l'ordre donné par **Comparable** on construit une classe qui implémente l'interface **Comparator**, qui permettra de comparer 2 éléments de la collection. Elle comporte une seule méthode **int compare(Object o1, Object o2)** qui doit renvoyer

- un entier positif si **o1** est « plus grand » que **o2**
- 0 si **o1** a la même valeur (au sens de **equals**) que **o2**
- un entier négatif si **o1** est « plus petit » que **o2**

```

//Critère de comparaison
public class CompareSalaire implements Comparator <Employee>{
    public int compare(Object o1, Object o2) {
        double s1 = ((Employee) o1).getSalaire();
        double s2 = ((Employee) o2).getSalaire();
        return (int)(s1 - s2);
    }
}
//..
List employes = new ArrayList();
// On ajoute les employés . . .
Collections.sort(employes,new CompareSalaire());
System.out.println(employes);

```

Depuis java 8, il est possible lorsqu'une interface définit une seule méthode on peut utiliser le Lambda Language.

```

Collections.sort(employes,new Comparator((Employee) o1, (Employee) o2)->{
    return (int)(o1.getSalaire()- o2.getSalaire());
});

```

b) Recherche

Pour rechercher la position d'un objet dans une liste on peut utiliser les méthodes suivantes :

- **int binarySearch(List l, Object obj)**
- **int binarySearch(List l, Object obj, Comparator c)**

La liste doit être triée en ordre croissant suivant l'ordre naturel (interface **Comparable**) ou suivant le comparateur, pour la 2ème méthode.

6. La gestion du temps (Date/Time)

Avant JDK 1.8, on avait quelques classes pour l'obtention et le formatage de la date répartie : `java.util.Date` (Date/Time), `java.util.Calendar` (Calendrier) , `java.text.DateFormat` et `java.text.SimpleDateFormat` (formatage de la date).

```
Date now = new Date();
System.out.println(now); // Date time Système

// Utilisation DateFormat
DateFormat formatter = DateFormat.getInstance(); // Date and time
String dateStr = formatter.format(now);
System.out.println(dateStr);
formatter = DateFormat.getTimeInstance(); // time only
System.out.println(formatter.format(now));

// Utilisation de locale (Propriétés locale au système)
formatter = DateFormat.getDateInstance(DateFormat.FULL,
    DateFormat.FULL, Locale.FRANCE);
System.out.println(formatter.format(now));

// Utilisation SimpleDateFormat
SimpleDateFormat simpleFormatter = new SimpleDateFormat("E yyyy.MM.dd 'at'
    hh:mm:ss a zzz");
System.out.println(simpleFormatter.format(now));
```

A partir de JDK 1.8, on a introduit l'api `java.time` qui concrétise toutes les fonctionnalités de la gestion du temps : `java.time.chrono.*`, `java.time.format.*`, `java.time.temporal.*`, `java.time.zone.*`

```
LocalDate today = LocalDate.now();
DayOfWeek dow = DayOfWeek.MONDAY;
LocalDate dateOfBirth = LocalDate.of(1972, Month.NOVEMBER, 23);
System.out.printf("%s\n", DayOfWeek.MONDAY.plus(3)); // JEUDI
System.out.printf("%d\n", Month.FEBRUARY.maxLength()); //29
YearMonth date = YearMonth.now(); // Mois de l année
boolean validLeapYear = Year.of(2012).isLeap(); //est c année bissextile
```

4. Les entrées sorties

1. Les flux java

Il existe plusieurs classes en java définies dans java.io qui permettent de traiter les flux d'entrées sorties qu'on peut classer en deux catégories :

- classes de flux d'octets : `InputStream` pour les entrées et `OutputStream` pour les sorties
- classes de flux de caractères : `Reader` pour les entrées et `Writer` pour les sorties

Il est à noter que le package java.io contient d'autres classes pour le filtrage des données, la concaténation de flux, la manipulation de fichiers, etc

2. La classe File

Cette classe fournit une définition *platform-independent* des fichiers et des répertoires : Un objet de type `File` peut représenter un fichier ou un répertoire. Elle permet de réaliser les opérations de base sur les fichiers tels que le listage d'un répertoire, la vérification de l'existence d'un fichier, etc.... Une partie des méthodes de la `File` sont illustrées ci-dessous :

Nom méthode	rôle
boolean canRead(), boolean canWrite() et boolean canExecute()	indique si le fichier peut être lu, écrit ou exécuté
File[] listRoots()	Retourne le disque associé au fichier sous Linux c'est / mais pour windows c'est C : ou D : ou autre
boolean delete()	détruire le fichier ou le répertoire. Le booléen indique le succès de l'opération
boolean exists()	indique si le fichier existe physiquement
String getAbsolutePath()	renvoie le chemin absolu du fichier
String getPath()	renvoie le chemin du fichier
boolean isDirectory()	indique si le fichier est un répertoire
boolean isFile()	indique si l'objet représente un fichier
long length()	renvoie la longueur du fichier
String[] list()	renvoie la liste des fichiers et répertoire contenu dans le répertoire

Depuis la version 1.2 du J.D.K., de nombreuses fonctionnalités ont été ajoutées à cette classe :

- la création de fichiers temporaires (`createNewFile`, `createTempFile`)

- la gestion des attributs "caché" et "lecture seule" (isHidden, isReadOnly)
- des méthodes qui renvoient des objets de type File au lieu de type String (getParentFile, getAbsolutePath, getCanonicalFile, listFiles)

Les principaux constructeurs sont :

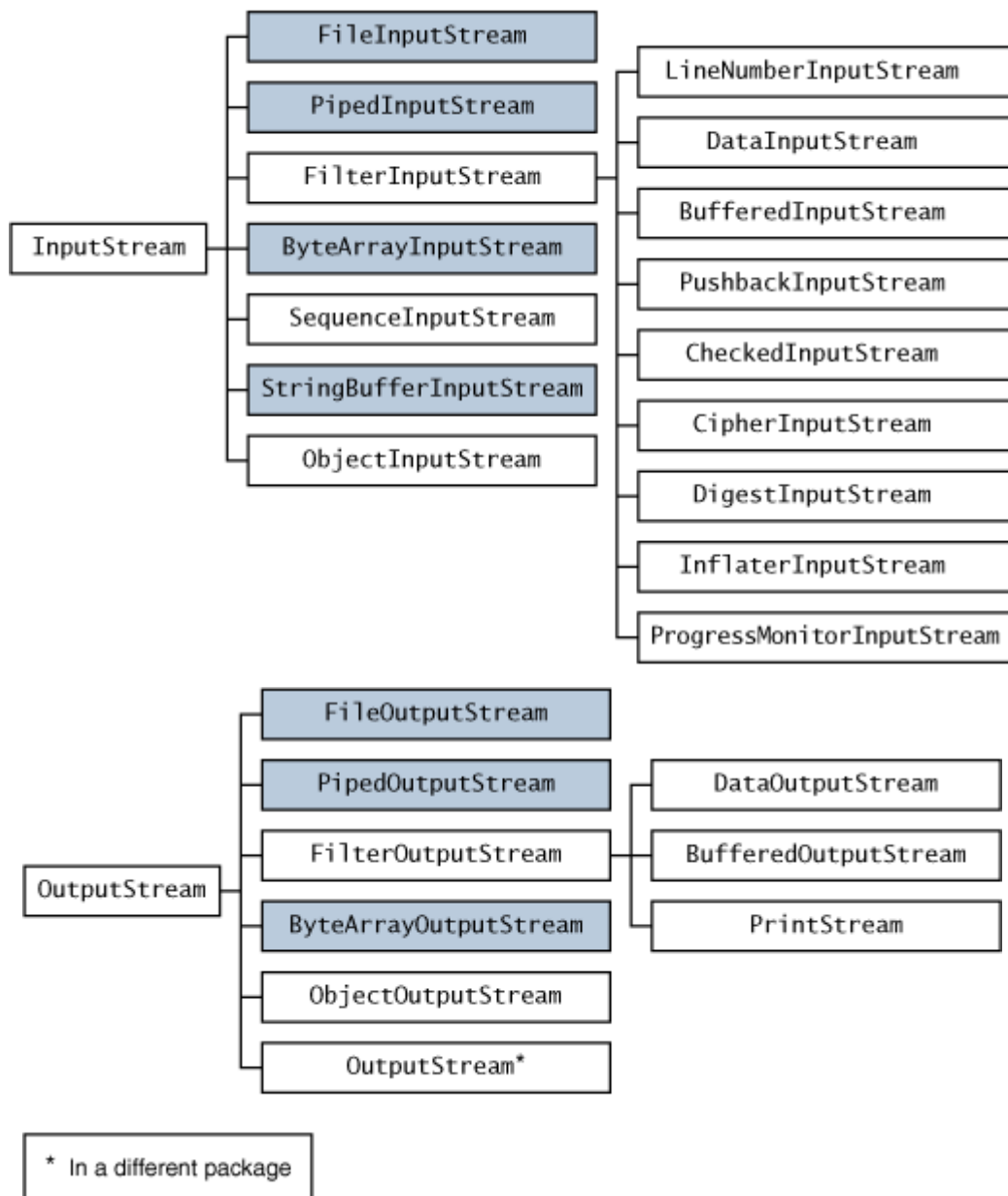
- File(File rep, String nom)
- File(String chemin)
- File(String rep, String nom)

```
public static void main(String[]args) {
Scanner in = new Scanner(System.in);
File fichier=new File(in.nextLine()); // Lire le nom d'un fichier
if (!fichier.exists()) {
System.out.println("le fichier "+nomFichier+"n'existe pas");
System.exit(1);
}
System.out.println("Nom du fichier      : "+fichier.getName());
System.out.println("Chemin du fichier : "+fichier.getPath());
System.out.println("Chemin absolu  : "+fichier.getAbsolutePath());
System.out.println("Droits : r("+fichier.canRead());
System.out.println("w( "+fichier.canWrite()+ ")");

if (fichier.isDirectory() ) {
System.out.println(" contenu du repertoire ");
File fichiers[] = fichier.listFiles();
for(int i = 0; i <fichiers.length; i++) {
if (fichiers[i].isDirectory())
System.out.println("d-"+fichiers[i].getName());
else
System.out.println("f-"+fichiers[i].getName());
}}}
}
```

3. Les flux octets

Ce sont des flux d'entrées sorties de base ils permettent de lire et écrire tout type de données. On se contentera dans ce cours par étudier les classes File(Input|Output)Stream, Data(Input|Output)Stream et Object(Input|Output)Stream.



1. File(Input|Output)Stream

Ces classes permettent d'accéder en lecture et en écriture à un fichier

```

import java.io.*;
public class CopyBytes {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("source.txt");
        File outputFile = new File("destination.txt");
        FileInputStream in = new FileInputStream(inputFile);
        FileOutputStream out = new FileOutputStream(outputFile);
        int c;
        while ((c = in.read()) != -1)
            out.write(c);
        in.close(); out.close();
    }
}

```

2. Data (Input|Output) Stream

Ce sont ces classes qui possèdent des écritures ou lectures de plus haut niveau que de simples octets (des entiers, des flottants, ...).

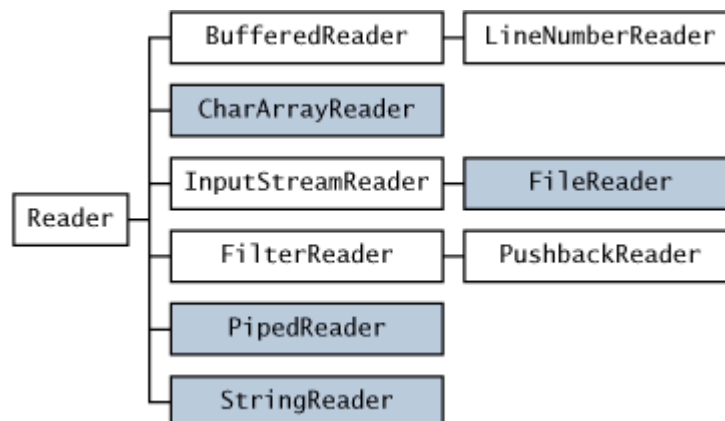
- `DataInputStream` ,
`public final boolean readBoolean() throws IOException`
`public final char readChar() throws IOException`
`public final int readInt() throws IOException`
...
- `DataOutputStream`,
`public final void writeBoolean(boolean) throws IOException`,
`public final void writeChar(int) throws IOException`,
`public final void writeInt(int) throws IOException`

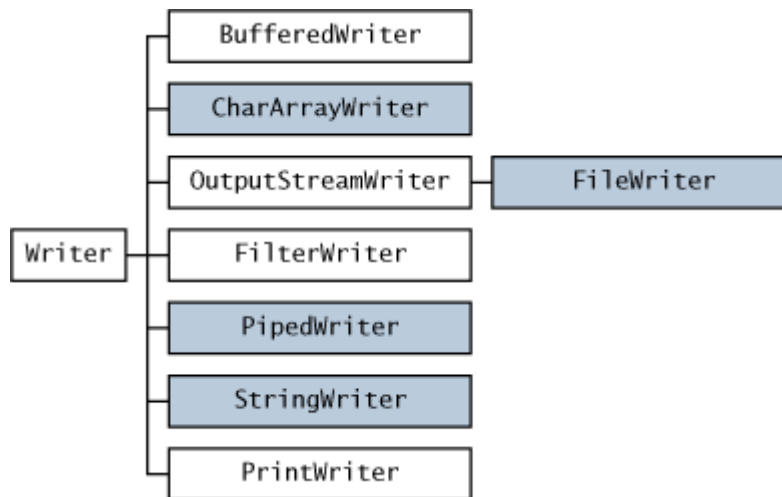
Si vous écrivez un entier avec `writeInt()` il faut le lire avec `readInt()`. Les chaînes de caractères sont lues par la méthode `readUTF()`. Pour lire un fichier texte ligne par ligne on peut utiliser la méthode `readLine()`. Cette dernière a été dépréciée et donc vous pouvez la remplacer par la méthode de `BufferedReader` comme suit :

```
File inputFile = new File("source.txt");
DataInputStream in = new DataInputStream(inputFile) ;
BufferedReader brin = new BufferedReader(in) ;
String L=Brin.readLine() ;
```

4. Les flux caractères

Ces classes permettent de faire des entrées sorties en mode caractères mais en unicode (2 octets).





```

import java.io.*;
public class Copy {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("source.txt");
        File outputFile = new File("destination.txt");

        FileReader in = new FileReader(inputFile);
        FileWriter out = new FileWriter(outputFile);
        int c;

        while ((c = in.read()) != -1)
            out.write(c);

        in.close();
        out.close();
    }
}

```

5. Les filtres

Un filtre permet de sélectionner les fichiers qui vérifient certains critères. En Java un filtre est un objet d'une classe qui implémente l'interface `FilenameFilter`. Cette classe doit donner une définition de la méthode `public boolean accept(File rep, String nom)` et seuls les noms de fichiers qui renvoie `true` par cette méthode seront affichés.

```

import java.io.*;
class Filtre implements FilenameFilter {
    public boolean accept(File rep, String nom) {
        if (nom.endsWith(".java")) return true;
        return false;
    }
}
//dans une classe de test
String nomFics[ ] = (new File(rep)).list(new Filtre());

```

6. Sérialisation des objets

En Java, seules les données (et les noms de leur classe ou type) sont sauvegardées (pas les méthodes ou constructeurs). Si un objet contient un champ qui est un objet, cet objet inclus est aussi sérialisé et on obtient ainsi un arbre (un graphe) de sérialisation d'objets.

Des objets de certaines classes ne peuvent pas être sérialisés : c'est le cas des `Threads`, des `FileXXXputStream`, ...

Pour indiquer que les objets d'une classe peuvent être persistants on indique que cette classe implémente l'interface Serializable.

```
Public class A implements Serializable{  
    // attributs sérializables  
    //...}
```

Certaines classes sont par défaut sérializable (String et Date). Les flux Objets en java (Object(Input|Output)Stream) permettent de lire et écrire des objets sérialisables.

```
// Ecriture  
FileOutputStream fos = new FileOutputStream("tmp");  
ObjectOutputStream oos = new ObjectOutputStream(fos);  
oos.writeObject("Today");  
oos.writeObject(new Date());  
oos.flush();  
// Lecture  
FileInputStream fis = new FileInputStream("tmp");  
ObjectInputStream ois = new ObjectInputStream(fis);  
String today = (String)ois.readObject();  
Date date = (Date)ois.readObject();
```

Par défaut, tous les champs d'un objets sont sérialisés (y compris private) Cela peut poser des problèmes de sécurité. Alors il existe 3 solutions possibles:

- Réécrire les méthodes writeObjet() et readObject()
- Utiliser le mot clé **transient** permet d'indiquer qu'un champs ne doit pas être sérialisé : private transient passwd;

6. Le graphique en Java

1. Graphique en Java

1. Paquetages de base

Une interface graphique est formée d'une ou plusieurs fenêtres qui contiennent divers composants graphiques (*widgets*) tels que

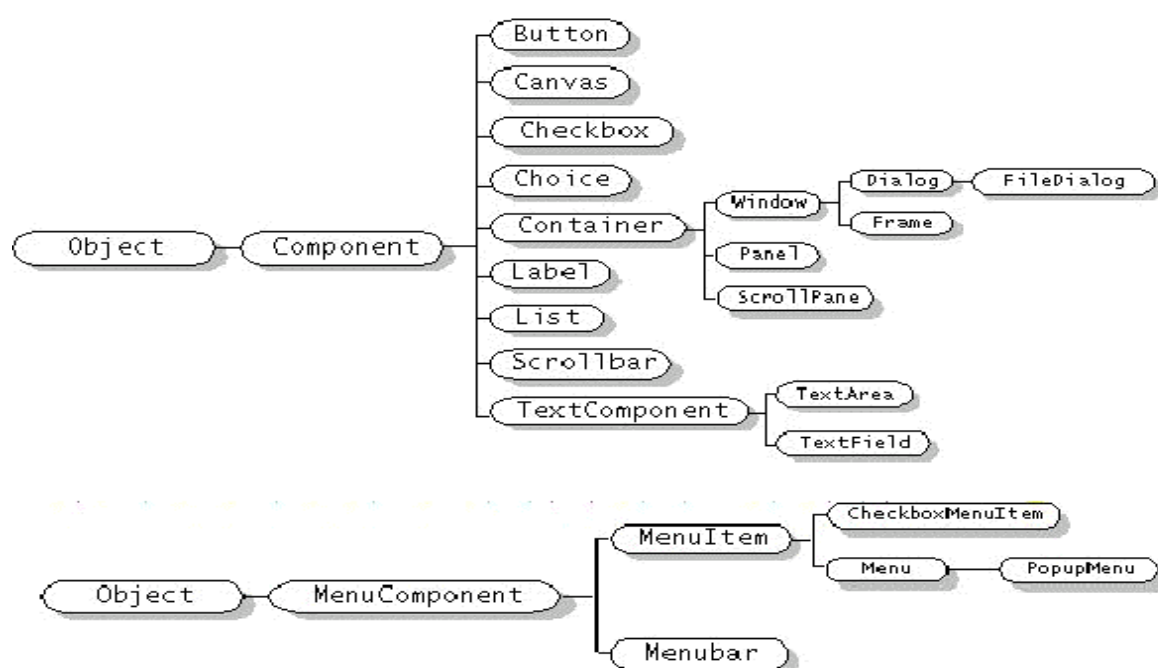
- boutons
- listes déroulantes
- menus
- champ texte, etc...

Les interfaces graphiques sont souvent appelés GUI d'après l'anglais *Graphical User Interface*. L'utilisation d'interfaces graphiques impose une programmation « conduite par les événements ». On peut distinguer deux packages de base pour la réalisation des GUI en java : `java.awt` (*Abstract Window Toolkit*, JDK 1.1) et `javax.swing` (JDK 1.2) qui font partie de JFC (*Java Foundation Classes*).

Techniquement Swing est construit au-dessus de AWT

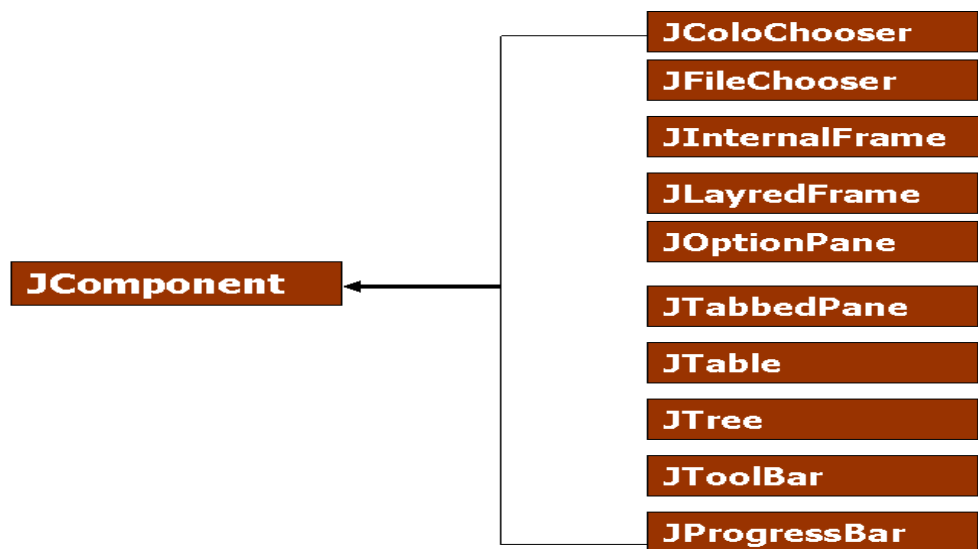
- même gestion des événements
- les classes de *Swing* héritent des classes de AWT

2. Les principaux composants awt



3. Les principaux composants swing

Tous les composants de AWT ont leur équivalent dans Swing en plus joli et avec plus de fonctionnalités. Syntaxiquement un objet swing est un objet awt avec un J comme préfixe ainsi une étiquette Label en awt devient un JLabel en swing. En plus Swing offre de nombreux composants qui n'existent pas dans AWT. La figure ci-dessous illustre quelques exemples de ces composants.



4. Premier programme Swing

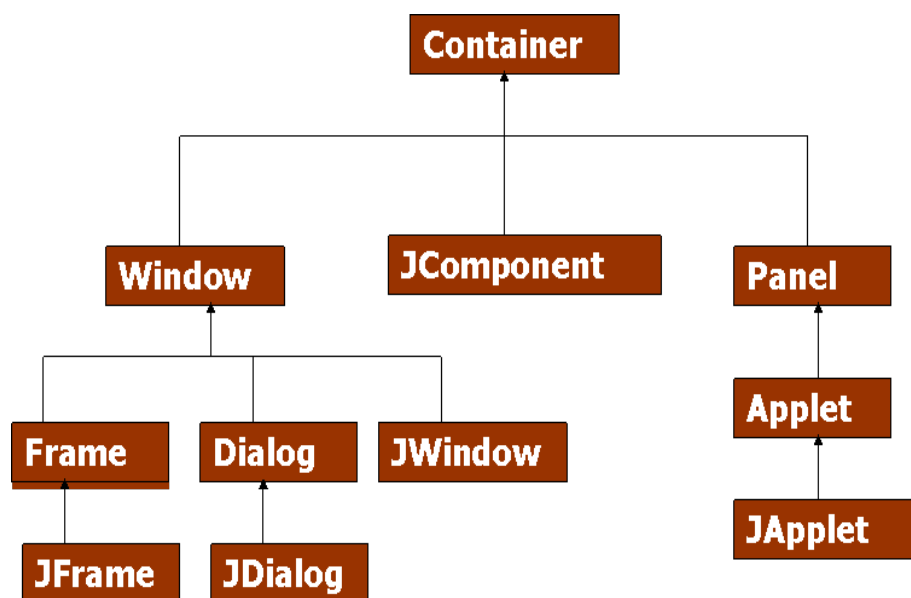
```
import javax.swing.*;
public class HelloWorldSwing {
    //Déclaration d'une fenêtre et d'une étiquette.
    JFrame frm;
    JLabel lbl;
    public void GUI() {
        //Création et initiation de la fenêtre.
        frm = new JFrame("HelloWorldSwing");
        //création d'une étiquette "Hello World".
        lbl = new JLabel("Hello World");
        //être sûr d'avoir une bonne décoration v1.4.
        JFrame.setDefaultLookAndFeelDecorated(true);
        //Fermer l'application dès qu'on clique sur le bouton fermer
        frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //ajouter le label à la fenêtre
        frm.add(lbl);
        //Ajuster la taille de la fenêtre à la taille de ses composants
        frm.pack();
        //Afficher la fenêtre.
        frm.setVisible(true); }
    }
    class Test{
    public static void main(String[] args) {
        ( new HelloWorldSwing()). GUI(); }
    }
```

2. Composants lourds/ légers

Pour afficher des fenêtres (instances de `JFrame`), Java s'appuie sur les fenêtres fournies par le système d'exploitation hôte dans lequel tourne la JVM, on dit que les `JFrame` sont des composants lourds. Au contraire de AWT qui utilise les *widgets* du système d'exploitation pour tous ses composants graphiques (fenêtres, boutons, listes, menus, etc.), Swing ne les utilise que pour les fenêtres de base « *top-level* ». Les autres composants, dits légers, sont dessinés dans ces containers lourds, par du code « pur Java »

1. Les conteneurs

Pour construire une interface graphique avec Swing, il faut créer un (ou plusieurs) container lourd et placer à l'intérieur les composants légers qui forment l'interface graphique.



Il y a 3 sortes de containers lourds (un autre, `JWindow`, est rarement utilisé) :

- `JFrame` fenêtre pour les applications
- `JApplet` pour les *applets*
- `JDialog` pour les fenêtres de dialogue

Il existe d'autres types de conteneurs qu'on appelle intermédiaires légers tels que **`JPanel`**, **`JScrollPane`**, **`JSplitPane`**, **`JTabbedPane`** qui servent à regrouper des composants dans une zone spécifique d'un conteneur lourd.

Les principales méthodes

Méthode	Rôle
<code>component add(Component comp)</code>	pour ajouter le composant spécifié à la fin du conteneur
<code>component add(Component comp, int index)</code>	Ajoute le composant spécifié à la position spécifiée par index (0...)

void remove(int index)	Supprime le composant situé à la position index
void remove(Component comp)	Supprime le composant spécifié
void removeAll()	Supprime tous les composants
void setLayout(LayoutManager mgr)	Définit le Layout (Mise en page) de ce conteneur
void paint(Graphics g)	Pour dessiner
void repaint()	Met à jour le dessin du conteneur

2. La fenêtre JFrame

Taille d'une fenêtre

La méthode **pack()** donne à la fenêtre la taille nécessaire pour respecter les tailles préférées des composants de la fenêtre. Sinon si vous voulez une taille ou un emplacement précis sur l'écran (en pixels) vous pouvez vous servir des méthodes suivantes :

- **setLocation(int xhg, int yhg)** (ou **Point** en paramètre)
- **setSize(int largeur, int hauteur)** (ou **Dimension** en paramètre)
- **setBounds(int x, int y, int largeur, int hauteur)** (ou **Rectangle** en paramètre)

La taille de l'écran ainsi que sa résolution en cours peuvent être obtenues en utilisant la classe abstraite **java.awt.Toolkit** implémenté dans AWT. Quelques méthodes publiques de cette classe sont : **getScreenSize**, **getScreenResolution**, **getDefaultToolkit**, **getImage**.

```
//Centrage de la fenêtre
Toolkit tk = Toolkit.getDefaultToolkit();
Dimension d = tk.getScreenSize();
int hauteurEcran = d.height;
int largeurEcran = d.width;
//Ajuster la taille et la position de la fenêtre.
frame.setSize(largeurEcran/2, hauteurEcran/2);
frame.setLocation(largeurEcran/4, hauteurEcran/4);
```

3. Composants de base

1. Etiquette JLabel

Avec un JLabel, on peut créer un texte « étiquette » et/ou une image. On peut également intégrer du HTML, spécifier la position du texte par rapport à l'image ou spécifier l'emplacement par rapport à son conteneur. Quelques méthodes publiques de cette classe :

- **JLabel(String, Icon, int)** // Création d'un JLabel
- **void setText(String)** // Modifie le texte du JLabel
- **String getText()** // retourne le texte du JLabel
- **void setIcon(Icon)** et **Icon getIcon()**
- // spécifier la position du texte par rapport à l'icone
 - **void setHorizontalTextPosition(int)**
 - **void setVerticalTextPosition(int)**

- `void setToolTipText(String)` //associe un info bulles

```
public class LabelPanel extends JPanel {
    public LabelPanel() {
        JLabel testLabel = new JLabel("Icon Big Label");
        testLabel.setToolTipText("a Label with Icone");
        // Créer une nouvelle fonte
        Font serif32Font = new Font("Serif",Font.BOLD|Font.ITALIC,32);
        // Donner une fonte au contenu du Label
        testLabel.setFont(serif32Font);
        // Créer une icone
        Icon soundIcon = new ImageIcon("images/sound.gif");
        // Placer 1 Icon dans le label
        testLabel.setIcon(soundIcon);
        // Aligner le Texte à droite de l'icone
        testLabel.setHorizontalTextPosition(JLabel.RIGHT);
        // Ajouter le label au panel
        add(testLabel);}}

```

2. Bouton JButton

Comme le JLabel, on peut créer un JButton avec un texte (bien évidemment HTML) et ou une icône.

```
//Création d'un Bouton avec icone
JButton precedent = new JButton("Precedent",leftButtonIcon);
//Création d'un Bouton avec texte HTML
new JButton("<html><h1>Ligne 1</h1></html>");
//positionnement du texte / à l'Icône
precedent.setVerticalTextPosition(AbstractButton.CENTER);
//Associer un raccourcis clavier au bouton ici p
precedent.setMnemonic(KeyEvent.VK_P);
//associer une action à un bouton
precedent.setActionCommand("disable");

```

Quelques méthodes utiles

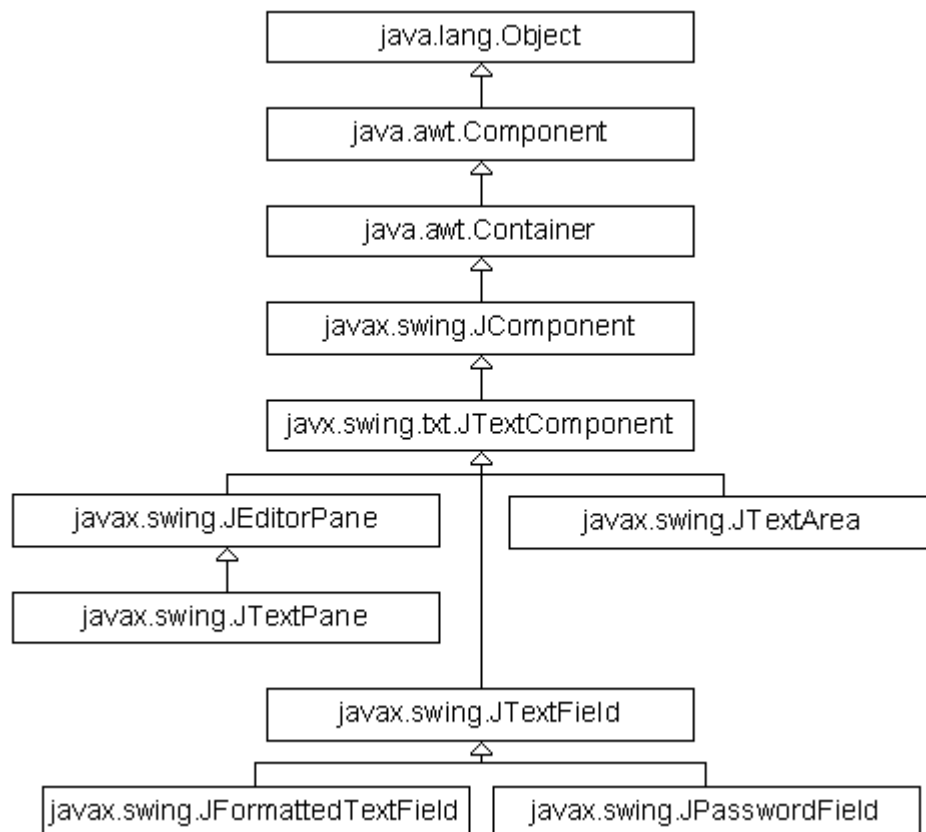
- `void setText(String)` et `String getText()`
- `void setIcon(Icon)`et `Icon getIcon()`
- `char getMnemonic()` `String getActionCommand()`
- `void setEnabled(Boolean)`

3. Les différents composants texte

La classe abstraite JTextComponent est la classe mère de tous les composants permettant la saisie de texte.

- **JTextField** : entrer d'une seule ligne
 - `JTextField tf= new JTextField();`
 - `tf.setText("nouveau Texte");`
- **JPasswordField** : entrer un mot de passe non affiché sur l'écran
 - `JPasswordField textfield = new JPasswordField (20);`
 - `textfield.setEchoChar('#');`/* par défaut
 - `char[] input = textfield.getPassword();`
- **JTextArea** : quelques lignes de texte avec une seule police de caractères
 - `textArea = new JTextArea(5, 40);`//lignes, colonnes
 - `append(String)` ajouter du texte à la fin

- **JTextPane** : documents avec plusieurs polices et des images et composants inclus (pas étudié dans cette partie du cours)



4. Mise en page des composants dans un conteneur

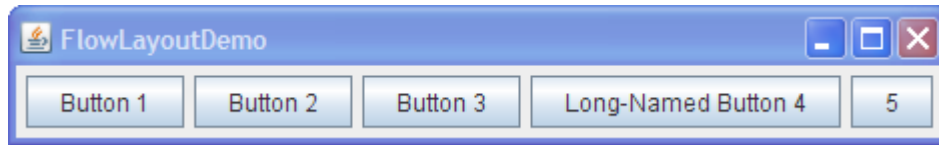
Si vous ajoutez plusieurs composants à un conteneur sans spécifier de Layout, seul le dernier sera visible. Les Layouts ou Layout Managers sont des classes qui implémentent une interface `LayoutManager`, et représentent des modèles de positionnement des composants. On peut citer comme exemples :

- **BorderLayout**
- **BoxLayout**
- **CardLayout**
- **FlowLayout**
- **GridBagLayout**
- **GridLayout**
- **GroupLayout**
- **SpringLayout**

1. FlowLayout

Quand un conteneur utilise `FlowLayout`, les composants sont arrangés dans l'ordre de leur ajout dans le conteneur, depuis le haut, de gauche à droite. On ne passe à la prochaine ligne que quand

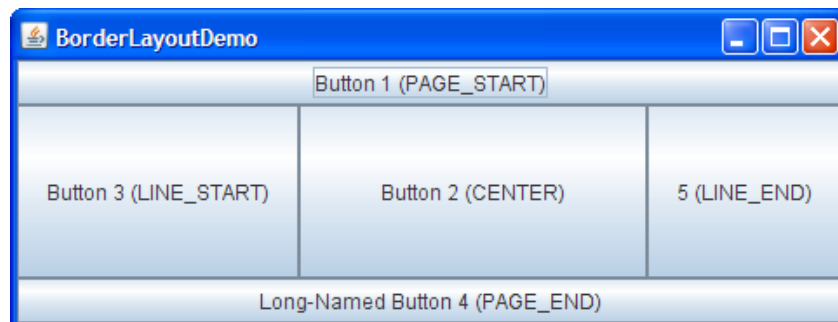
l'espace restant sur la ligne n'est plus suffisant pour contenir le composant. Il est utilisé par défaut dans les applets.



```
FlowLayout FLayout = new FlowLayout();
JFrame frm =new JFrame("FlowLayoutDemo");
...
    frm.setLayout(FLayout);
    frm.add(new JButton("Button 1"));
    frm.add(new JButton("Button 2"));
    frm.add(new JButton("Button 3"));
    frm.add(new JButton("Long-Named Button 4"));
    frm.add(new JButton("5"));
```

2. BorderLayout

Quand on ajoute un composant à un conteneur qui utilise BorderLayout, on spécifie la région dans laquelle on veut le positionner. Si la région n'est pas spécifiée, il le positionne automatiquement dans le centre. Les 5 régions utilisés par un BorderLayout sont :*NORTH*, *WEST*, *CENTER*, *EAST* et *SOUTH*



```
JButton button = new JButton("Button 1 (PAGE_START)");
JPanel pane=new JPanel("BorderLayoutDemo");
pane.add(button, BorderLayout.NORTH);
button = new JButton("Button 2 (CENTER)");
button.setPreferredSize(new Dimension(200, 100));
pane.add(button, BorderLayout.CENTER);
button = new JButton("Button 3 (LINE_START)");
pane.add(button, BorderLayout.WEST);
button = new JButton("Long-Named Button 4 (PAGE_END)");
pane.add(button, BorderLayout.SOUTH);
button = new JButton("5 (LINE_END)");
pane.add(button, BorderLayout.EAST);
```

3. GridLayout

Le GridLayout arrange les composants dans un nombre donné de colonnes et de lignes. Il commence en haut à gauche, et passe à la colonne suivante, et ainsi de suite, jusqu'à la dernière

colonne, puis passe à la deuxième ligne de la première colonne, ... etc. Le nombre de colonnes et de lignes est spécifié dans les arguments du constructeur :

new GridLayout(nombre_ligne, nombre_colonnes)

Le nombre de colonnes et de lignes ne peuvent pas être tous les deux des zéros. Si l'un ou l'autre est un zéro, il est interprété comme à déterminer par le nombre des composants.



```
JPanel p = new JPanel();  
p.setLayout(new GridLayout(3,2));  
p.add(new JButton("1"));  
p.add(new JButton("2"));  
p.add(new JButton("3"));  
p.add(new JButton("4"));  
p.add(new JButton("5"));  
p.add(new JButton("6"));
```

5. Traitement des événements

1. principe

L'utilisateur utilise le clavier et la souris pour intervenir sur le déroulement du programme. Le système d'exploitation engendre des événements à partir des actions de l'utilisateur ensuite le programme doit lier des traitements à ces événements. On distingue deux types d'événement

- Bas Niveau :
 - appui sur un bouton de souris ou une touche du clavier
 - relâchement du bouton de souris ou de la touche
 - déplacer le pointeur de souris
- Logique
 - frappe d'un A majuscule
 - clic de souris
 - choisir un élément dans une liste

Le JDK utilise une architecture de type « observateur - observé ». Les composants graphiques (comme les boutons) sont les observés. Chacun de ces composants a ses observateurs (ou écouteurs, *listeners*). Ces écouteurs sont prévenus par le composant graphique dès qu'un événement qui les concerne survient sur ce composant. Le code de ces écouteurs exécute les actions à effectuer en réaction à l'événement.

2. Classes d'événements

3.

Evénement, Interface de l'événement et Composants supportant cet événement
méthodes d'ajout et de suppression associées

Ecouteur	Composant
ActionEvent ActionListener addActionListener() removeActionListener()	JButton, JList, JTextField, JMenuItem and its derivatives including JCheckBoxMenuItem, JMenu, and JPopupMenu.
FocusEvent FocusListener addFocusListener() removeFocusListener()	Component and derivatives*.
KeyEvent KeyListener addKeyListener() removeKeyListener()	Component and derivatives*.
MouseEvent (for both clicks and motion) MouseListener addMouseListener() removeMouseListener()	Component and derivatives*.
MouseEvent (for both clicks and motion) MouseMotionListener addMouseMotionListener() removeMouseMotionListener()	Component and derivatives*.
ItemEvent ItemListener addItemListener() removeItemListener()	JCheckBox, JCheckBoxMenuItem, JComboBox, JList, and anything that implements the ItemSelectable interface
WindowEvent WindowListener addWindowListener() removeWindowListener()	Window and its derivatives, including JDialog, JFileDialog, and JFrame.

3. Ecouteur vs Adaptateur

Pour éviter au programmeur d'avoir à implanter toutes les méthodes d'une interface « écouteur », AWT fournit des classes (on les appelle des adaptateurs), qui implémentent toutes ces méthodes. Le code des méthodes ne fait rien. Ça permet au programmeur de ne redéfinir dans une sous-classe que les méthodes qui l'intéresse.

Les classes suivantes du paquetage **java.awt.event** sont des adaptateurs : **KeyAdapter**, **MouseAdapter**, **MouseMotionAdapter**, **FocusAdapter**, **ComponentAdapter**, **WindowAdapter**.

Supposons que vous voulez traiter le clique de la souris donc il suffit d'implémenter le traitement de la méthode `mouseClicked` comme suit :

```
addMouseListener(new MouseAdapter() {  
    public void mouseClicked(MouseEvent e) {  
        System.exit(0);  
    }  
});
```

Mais si vous utilisez un Listener au lieu d'un Adapter il faut implémenter toutes les méthodes de l'interface `MouseListener`.

Listener vs. adapter	Methods in interface
ActionListener	<code>actionPerformed(ActionEvent)</code>
AdjustmentListener	<code>adjustmentValueChanged(AdjustmentEvent)</code>
ContainerListener	<code>componentHidden(ComponentEvent)</code>
ContainerAdapter	<code>componentShown(ComponentEvent)</code> <code>componentMoved(ComponentEvent)</code> <code>componentResized(ComponentEvent)</code>
FocusListener	<code>focusGained(FocusEvent)</code>
FocusAdapter	<code>focusLost(FocusEvent)</code>
KeyListener	<code>keyPressed(KeyEvent)</code>
KeyAdapter	<code>keyReleased(KeyEvent)</code> <code>keyTyped(KeyEvent)</code>
MouseListener	<code>mouseClicked(MouseEvent)</code>
MouseAdapter	<code>mouseEntered(MouseEvent)</code> <code>mouseExited(MouseEvent)</code> <code>mousePressed(MouseEvent)</code> <code>mouseReleased(MouseEvent)</code>
MouseMotionListener	<code>mouseDragged(MouseEvent)</code>
MouseMotionAdapter	<code>mouseMoved(MouseEvent)</code>

4. Exemple Complet sur ActionEvent

Cette classe décrit des événements de haut niveau très utilisés qui correspondent à un type d'action de l'utilisateur qui va le plus souvent déclencher un traitement (une *action*) :

- clic sur un bouton
- *return* dans une zone de saisie de texte
- choix dans un menu

Ces événements sont très fréquemment utilisés et ils sont très simples à traiter. Un objet *ecouteur* intéressé par les événements de type « action » (classe **ActionEvent**) doit appartenir à une classe qui implémente l'interface **java.awt.event.ActionListener**. L'exemple suivant affiche le nombre de clique sur le bouton ok dans le bouton lui-même.

```
//Fenetre est un écouteur de ActionEvent
class Fenetre extends JFrame implements ActionListener{
int nbClique=0;
public Fenetre(){
JButton ok=new JButton(' 'OK' '+nbClique);
//inscription du bouton ok auprès de l'écouteur
Ok.addActionListener(this);
getContentPane().add(ok);}

//message déclenché lorsque l'utilisateur clique sur le bouton
public void actionPerformed(ActionEvent e){
nbClique++;
Object source = e.getSource();
If (Object==ok) Ok.setText(' 'OK' '+nbClique);
}}
```

Cette façon de faire est un peu lourde c'est pourquoi on utilise généralement un écouteur anonyme de la façon suivante :

```
class Fenetre extends JFrame{
int nbClique=0;
public Fenetre(){
JButton ok=new JButton(' 'OK' '+nbClique);
//inscription du bouton ok auprès de l'écouteur
ok.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {
nbClique++;
Ok.setText(' 'OK' '+nbClique);
} });}}
```

Depuis java 8, il est possible lorsqu'une interface définit une seule méthode d'utiliser le Lambda Langage .

```
ok.addActionListener(e ->{nbClique++;
Ok.setText(' 'OK' '+nbClique);});
```

6. Les autres Composants graphiques

JCheckBox zone à cocher

Une zone à cocher Java est traitée en tant que bouton et elle peut être intégrée à un menu en utilisant la classe JCheckBoxMenuItem.

```
//Création d'un checkBox avec icône
JCheckBox cbn= new JCheckBox("No Choose Me", false);
JCheckBox cbc= new JCheckBox("Choose Me");
//selectionner cbn par défaut
cbc.setSelected(true);
//recuperer l etat d un checkbox
boolean etat=cbc.isSelected()
// associer un raccourcis clavier
cbc.setMnemonic(c);
//Ecouteur JCheckBox
```

```
cbc.addItemListener(new ItemListener(){
public void itemStateChanged(ItemEvent e)
{
    // faire quelques choses}}) ;
```

JRadioButton

Dans awt, les boutons radio sont des checkboxes qui appartiennent au même CheckboxGroup. Swing offre un nouveau composant qui est JRadioButton et qu'on peut est ajouté à un ButtonGroup. Comme CheckboxGroup, le ButtonGroup est un composant logique qui n'a aucun effet visuel

```
ButtonGroup Payement = new ButtonGroup();
JRadioButton btrCheque = new JRadioButton("Chèque");
JRadioButton btrEspece = new JRadioButton("Espèce");
// ajouter le bouton au groupe
Payement.add (btrCheque);
Payement.add (btrEspece);
btrEspece.setSelected(true); btrEspece.setMnemonic(E);
```

JComboBox Zone de Liste modifiable

Même fonctionnement que le composant Choice avec quelques comportements supplémentaires. Il permet à un utilisateur de faire un ou plusieurs choix à la fois. La forme par défaut d'un JComboBox est non éditable.

```
//créer un combobox
String[] Villes = {"Rabat", "Casa", "Oujda", "Layoune"};
JComboBox cbVilles= new JComboBox() ;
//Ajouter les éléments à la liste
for (int i=0;i<Villes.length;i++) { cbVilles.addItem (Villes[i]);}
// On peut initialiser la liste au moment de sa création
JComboBox cbVilles= new JComboBox(Villes) ;
// choisir un element par default
cbVilles.setSelectedItem("Rabat");//setSelectedIndex(0)
```


Méthodes	Rôles
insertItemAt(Objetc, int)	Ajouter / Récupérer un élément dans une position
Object getItemAt(int) ou	spécifique ou sélectionné
Object getSelectedItem()	
removeAllItems() removeItemAt(int)	Supprimer un ou plusieurs éléments
void removeItem(Object)	
setEditable(Boolean) isEditable()/	Modifier l'état de la Combo et récupérer le nombre
getItemCount()	d'éléments
ActionListener	Les écouteurs possibles
ItemListener	

Liste non modifiable JList

Une JList permet d'afficher des valeurs sous forme d'une ou plusieurs colonnes et est généralement intégrée dans une JScrollPane. Une JList est un composant qui implémente le Modèle MVC à l'instar de JTable, JTree,...

- Création d'une JList

```
JList list = new JList(data); //data de type Object[]
```



```
//Pour sélectionner un élément à la fois
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION
);

//Pour sélectionner plusieurs éléments consécutifs
list.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_
SELECTION);

//Pour sélectionner plusieurs éléments
list.setSelectionMode(ListSelectionModel.
MULTIPLE_INTERVAL_SELECTION);
```

- Quelques méthodes

Le plus important dans une JList est de pouvoir récupérer l'élément ou les éléments sélectionnés. Pour cela plusieurs méthodes sont disponibles :

Méthodes	Rôles
int getSelectedIndex()	Récupérer l'indexe de l'élément sélectionné
int getMinSelectionIndex()	Récupérer l'indexe minimal est maximal de la sélection
int getMaxSelectionIndex()	
int[] getSelectedIndices()	Récupérer tous les indexes sélectionnés dans un tableau en ordre croissant
Object getSelectedValue()	Récupérer l'objet sélectionné au lieu de l'indice, n'oublier pas de faire le cast
Object[] getSelectedValues()	Récupérer les objets sélectionnés dans un tableau en ordre croissant

L'utilisation standard d'une liste est de demander à l'utilisateur de cliquer sur un bouton lorsqu'il a fini de faire ses choix dans la liste. Par ailleurs Il est rare d'écrire un écouteur de liste et on récupère alors la sélection de l'utilisateur par une des méthodes présentées ci-dessus. Sinon, La classe d'un écouteur de liste doit implémenter l'interface **ListSelectionListener** qui contient la méthode **void valueChanged(ListSelectionEvent e)**.

- **Modèle de JList**

On peut créer une JList en l'associant à un modèle qui fournit les données affichées par la liste : **public JList(ListModel dataModel)**. **ListModel** est une interface dont le corps est ci-dessous :

```
public interface ListModel {
    int getSize();
    Object getElementAt(int i);
    void addListDataListener(ListDataListener l);
    void removeListDataListener(ListDataListener l);}
```

Pour écrire une classe qui implémente **ListModel**, le plus simple est d'hériter de la classe abstraite **AbstractListModel** qui implémente les 2 méthodes de **ListModel** qui ajoutent et enlèvent les écouteurs. Les plus simples ont un modèle de la classe **DefaultListModel** qui hérite de la classe **AbstractListModel** et offre des méthodes d'ajout, de suppression et de récupération des éléments de la JList.

Exemple

```
ListModel pays = new DefaultListModel();
pays.addElement("Maroc");
pays.addElement("France");
pays.addElement("Italie");
pays.addElement("Espagne");
JList liste = new JList(pays);
```

- **Création d'un nouveau modèle**

Pour créer un nouveau modèle de JList il suffit d'étendre la classe **AbstractListModel** et d'implémenter les méthodes dont vous avez besoins.

```
import java.util.*;
class ProduitModel extends AbstractListModel
{
    public Vector produits;
    public ProduitModel()
    {produits = new Vector();}
    public int getSize(){return produits.size();}
    public Object getElementAt(int j)
    {return produits.elementAt(j);}
    public void addElement(Object a)
    {produits.addElement(a);
     this.fireIntervalAdded(this,0,getSize()-1);
    }}
}
```

La méthode **fireIntervalAdded** permet de rafraîchir le contenu de la JList en intégrant le nouveau produit ajouté.

Les boîtes de messages

JOptionPane est un composant léger, classe fille de **JComponent** elle permet d'avoir très simplement les cas les plus fréquents de fenêtres de dialogue message d'information avec bouton OK (**showMessageDialog**) demande de confirmation avec boutons Oui, Non et Cancel (**showConfirmDialog**) possibilité de configurer les boutons avec **showOptionDialog** saisie

d'une information sous forme de texte, **showInputDialog**. **JDialog** (composant lourd) est utilisée pour les cas non prévues par **JOptionPane**.

Les arguments complets de **JOptionPane** sont :

- Component frame // associée à la boîte de dialogue
- Object message // message à afficher
- String titre // le titre de la boîte de dialogue
- int TypeBouton // spécifier les boutons à faire apparaître
- int TypeIcône // détermine l'icône à afficher
- Object[] options // le message à afficher sur chaque bouton
- Object initialValue // le bouton sélectionné par défaut

Pour afficher un message d'erreur :

```
JOptionPane.showMessageDialog(null, "Erreur de saisie", "alert",  
JOptionPane.ERROR_MESSAGE);
```

Pour afficher un message de confirmation avec deux boutons oui/non par défaut

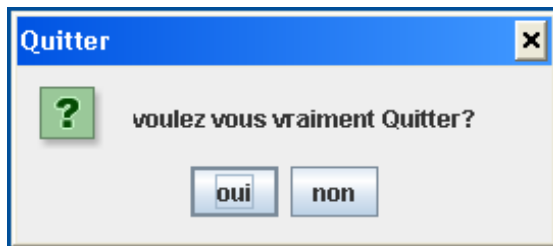
```
JOptionPane.showConfirmDialog(null,  
"Voulez vous vraiment Quitter", "Quitter", JOptionPane.YES_NO_OPTION);
```

On peut indiquer un type de message qui indiquera l'icône affichée en haut, à gauche de la fenêtre (message d'information par défaut)

- **JOptionPane.INFORMATION_MESSAGE**
- **JOptionPane.ERROR_MESSAGE**
- **JOptionPane.WARNING_MESSAGE**
- **JOptionPane.QUESTION_MESSAGE**
- **JOptionPane.PLAIN_MESSAGE** (pas d'icône)

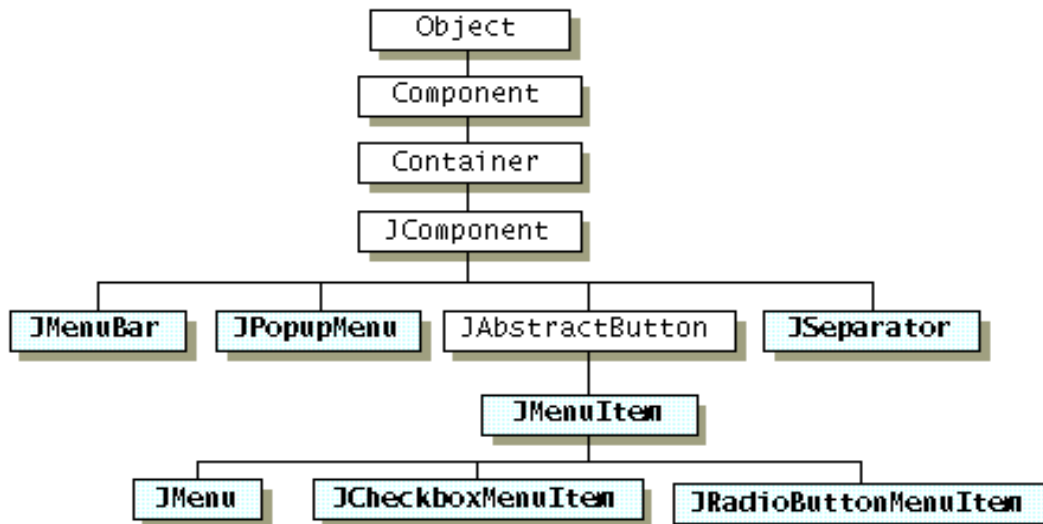
Les types de boutons à afficher dépendent de la méthode appelée :

- **showMessageDialog** : bouton Ok
- **showInputDialog** : Ok et Cancel
- **showConfirmDialog** : dépend du paramètre passé ; les différentes possibilités sont :
 - **DEFAULT_OPTION**
 - **YES_NO_OPTION**
 - **YES_NO_CANCEL_OPTION**
 - **OK_CANCEL_OPTION**
- **showOptionDialog** : selon le tableau d'objet passé en paramètres (vous pouvez franciser le texte à afficher sur les boutons)



```
String [] options={"oui","non"};
int n =
OptionPane.showOptionDialog(this,
    "voulez vous vraiment Quitter?","Quitter",
    JOptionPane.YES_NO_OPTION,OptionPane.QUESTION_MESSAGE,
    null,options,options[0]);
if(n==1) this.dispose();
```

Les Menus



Le modèle d'utilisation des menu swing est à peut près identique à celui des awt. Les classes de menu (**JMenuItem**, **JCheckboxMenuItem**, **JMenu**, et **JMenuBar**) sont toutes des sous classe de Jcomponent. Donc on peut placer un JMenuBar dans conteneur par **setMenuBar()**. On peut associer une icône à un JMenuItem. **JPopupMenu** ou Les menus pop-up sont des menus qui apparaissent à la demande (click bouton droit de la souris)

Exemple de construction de Menu

On va créer un menu Fichier avec trois options (nouveau, ouvrir et fermer).

```
//Création d'une barre de menu
JMenuBar jmb = new JMenuBar();
//Création d'un menu fichier
JMenu fichier = new JMenu ("Fichier");
fichier.setMnemonic(KeyEvent.VK_F);
//Ajouter des éléments au menu Fichier
JMenuItem itemNew,itemOpen,itemClose;
itemNew = new JMenuItem ("Nouveau");
itemOpen = new JMenuItem ("Ouvrir");
itemClose =new JMenuItem ("Fermer");
fichier.add (itemNew);
fichier.add (itemOpen);
//Ajout d'un séparateur de menu
fichier.addSeparator();
```

```
fichier.add(itemClose) ;
//Ajouter le menu fichier à la barre de menu
jmb.add (fichier);
//Ajouter le menu à la fenêtre
frame.setJMenuBar (theJMenuBar) ;
```

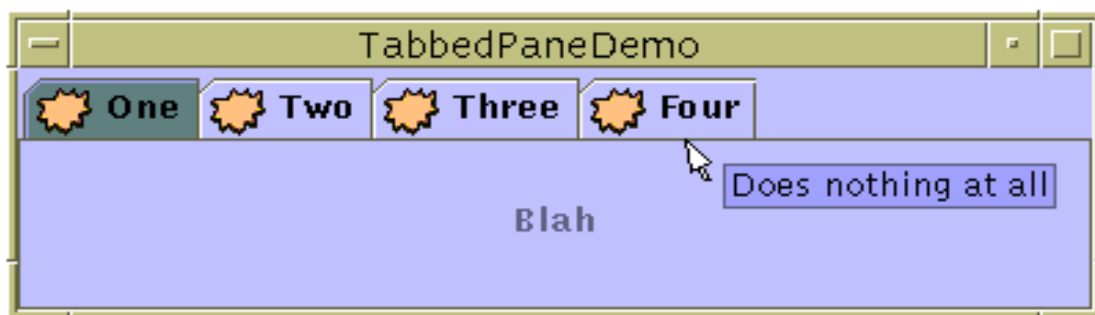
Pour traiter le clique sur un MenuItem il suffit d'implémenter l'écouteur ActionListener.

Exemple de JToolBar

```
//Création d'une barre d'outils
JToolBar toolbar = new JToolBar();
//Ajouter un élément à la barre d'outils
JButton ouvrir=new JButton();
ouvrir.setIcon("images/open.gif");
toolbar.add(ouvrir);
//Désactiver le déplacement de la barre d'outils
toolbar.setFloatable (false);
```

7. JTabbedPane (Onglets)

JTabbedPane permet à un utilisateur de switcher parmi plusieurs onglets par clique. Chaque onglet est composé d'un titre, d'un pannel et éventuellement une image. L'ajout des onglets au TabbedPane est effectué par la méthode addTab ou insertTab. Chaque onglet disposé d'un indice correspondant à la position dont laquelle il a été ajouté, le premier est indexé par 0 et le dernier par n-1 ou n est le nombre d'onglets ajoutés



```
JTabbedPane tabbedPane = new JTabbedPane();
ImageIcon icon = createImageIcon("images/middle.gif");

JComponent panel1 = makeTextPanel("Panel #1");
tabbedPane.addTab("Tab 1", icon, panel1, "Does nothing at all");
tabbedPane.setMnemonicAt(0, KeyEvent.VK_1);
```

Les methods de JtabbedPane

```
// ajouter supprimer un élément
insertTab(String, Icon, Component, String, int)
void removeTabAt(int) void removeAll()
//Récupérer un élément
int indexOfComponent(Component|String|Icon)
int getSelectedIndex() Component getSelectedComponent()
//changer la couleur de l'arrière plan et du style d'écriture
```

```
void setBackgroundAt(int, Color) Color getBackgroundAt(int)
void setForegroundAt(int, Color) Color getForegroundAt(int)
//Activer désactiver un onglet
void setEnabledAt(int, boolean) boolean isEnabledAt(int)
```

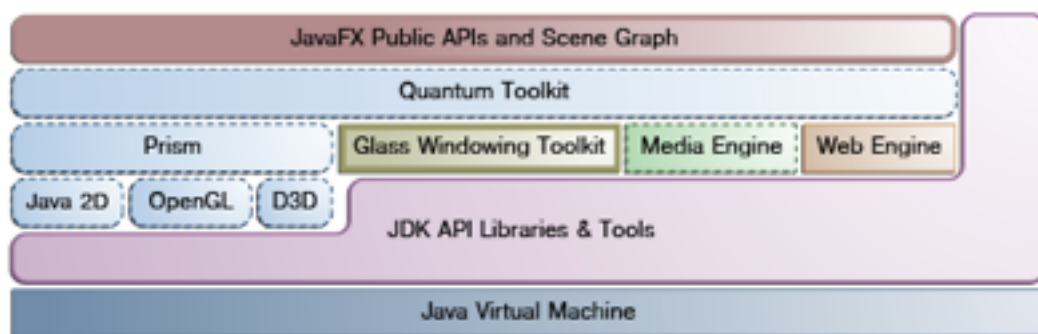
7. JavaFX

1. Introduction

À l'origine, la plate-forme JavaFX était principalement destinée aux applications Internet riches (RIA). En 2014, JavaFX a été intégré dans JSDK 8 en tant que nouveauté sous forme d'API de version 8. Depuis Java 11, JavaFX a été supprimé du SDK Java pour devenir un projet open source OpenJFX (<https://openjfx.io>). Maintenant JavaFX est une plate-forme Open Source de nouvelle génération pour le développement d'applications clientes riches : Bureau, Web et Mobiles / Embarqués basée sur Java.

2. Architecture

JavaFX est un ensemble de bibliothèques Java conçues pour permettre aux développeurs de créer et de déployer des applications client riches qui se comportent de manière cohérente sur toutes les plateformes.



Scene graph

Le **JavaFX scene graph**, présenté dans la couche supérieure de la figure 1, constitue le point de départ de la création d'une application JavaFX. C'est une arborescence hiérarchique de nœuds qui représente tous les éléments visuels de l'interface utilisateur de l'application. Chaque nœud a un identifiant, une classe de style et supporte un ensemble d'événements. À l'exception du nœud racine d'un graphe de scène, chaque nœud a un seul parent et zéro enfant ou plus.

Prism

Prism est un moteur haute performance permettant l'accélération matérielle utilisé pour restituer les graphiques au format 2D et 3D.

Dans l'absence de l'accélération sur le matériel ou le GPU, Prism utilise DirectX sur Windows et OpenGL sur Mac, Linux, et les systèmes embarqués.

Glass

Glass Windowing, représente le niveau le plus bas de la pile de graphiques JavaFX. Sa

responsabilité principale consiste à fournir des services du Système d'exploitation natifs, tels que la gestion des fenêtres, des minuteries et des surfaces. C'est l'intermédiaire entre la plate-forme JavaFX et le système d'exploitation natif. Glass est également responsable de la gestion de la file d'attente d'événements. Contrairement à (AWT), qui gère sa propre file d'attente d'événements, Glass utilise les fonctionnalités de file d'attente des événements du système d'exploitation natif pour planifier l'utilisation des processus.

Quantum Toolkit

Quantum Toolkit relie Prism et Glass Windowing Toolkit et les met à la disposition de la couche JavaFX située au-dessus d'eux dans la pile. Il gère également les règles de thread liées au rendu par rapport à la gestion des événements.

WebEngine : javafx.scene.web

Basé sur l'API open source web kit, WebEngine permet à une application javaFX d'interpréter un contenu HTML (HTML5, CSS, JavaScript, DOM and SVG)

MediaEngine : javafx.scene.media

Il est basé sur un moteur open source appelé Streamer. Ce moteur multimédia prend en charge la lecture de contenu vidéo et audio. Il supporte les formats mp3, wav, aiff et flv.

3. Application JavaFX

Introduction

JavaFX vous permet de concevoir avec MVC (Model-View-Controller) en utilisant FXML (extension XML pour JavaFX) et Java. Le "Modèle" est constitué d'objets du domaine spécifiques à l'application, la "Vue" est constituée de FXML et le "Contrôleur" est un code Java

qui définit le comportement de l'interface graphique pour interagir avec l'utilisateur. La personnalisation de l'apparence et le style de l'application peut être fait via le CSS.

CDI : Context Dependency Injection permet de référencer un objet FXML depuis Java (Contrôleur).

Il est possible de concevoir la GUI sans écrire de code grâce à JavaFX Scene Builder. Lors de la conception de l'interface utilisateur, Scene Builder crée un balisage FXML pouvant être transféré dans un environnement de développement intégré (IDE) afin que les développeurs puissent ajouter la logique métier.

Cycle de vie d'une application JavaFX

Une application JavaFX hérite de `javafx.application.Application`. La JVM maintient le cycle de vie d'une application JavaFX comme suit:

- Il construit une instance de Application en utilisant la méthode `launch()`
- Il appelle la méthode `init()` de l'application.
- Il appelle la méthode `start` de l'application (`javafx.stage.Stage`) et passe une instance de Stage en argument.
- Il attend la fin de l'application (par exemple, via `Platform.exit()` ou en fermant toutes les

fenêtres).

- Il appelle la méthode `stop ()` de l'application.

`Start ()` est une méthode abstraite, qui doit être redéfinie. `init()` et `stop()` ont une implémentation par défaut qui ne fait rien.

Si vous utilisez `System.exit (int)` pour mettre fin à l'application, `stop ()` ne sera pas appelé.

Structure d'une application JavaFX

Une application graphique javaFX (`javafx.application.Application`) est composée de :

Stage (`javafx.stage.Stage`) : C'est le conteneur racine, il représente une fenêtre qu'on associe à une scène. Il est passé comme argument de la méthode `start` **public void start(Stage primaryStage)**

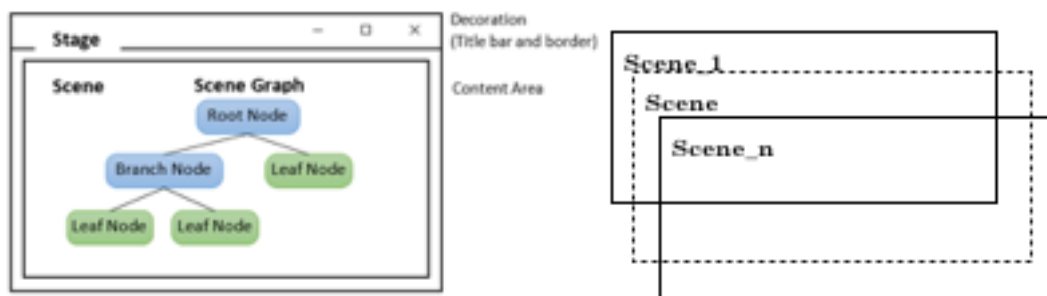
Scene (`javafx.scene.Scene`) : C'est le conteneur des composants graphiques (Scene Graph). Une application peut avoir plusieurs scènes, mais une seule de ces scènes peut être affichée dans un Stage à un instant donné.

Pour construire un objet Scene, on a besoin d'un noeud racine (Parent), la largeur et la hauteur :

Scene(Parent root, double width, double height)

Scene Graph (hierarchie de Noeuds) : Un noeud est une classe abstraite (`javafx.scene.Node`) super-class de tous les éléments graphiques (UI) :

- **Controls** (Composants): sous-classes de Parent dans le package `javafx.scene.control`, par exemple Label, TextField, Button.
- **Layout Pane** (Conteneurs): sous-classes de Parent dans le package `javafx.scene.layout`, par exemple, StackPane, FlowPane, GridPane, BorderPane.
- **Geometrical Shapes**: sous-classes de Shape et Shape3D dans le package `javafx.scene.shape`, par exemple, Circle, Rectangle, Polygon, sphere, Box.
- **Media Elements**: ImageView, MediaView (pouvant être lu par un lecteur multimédia) dans les packages `javafx.scene.image` et `javafx.scene.media`.



Un noeud implémente deux interfaces :

- `javafx.css.Styleable`: pour appliquer le style CSS.
- `javafx.event.EventTarget`: pour les événements

Les Layouts

FlowPane : C'est le FlowLayout swing, les composants sont insérés horizontalement ou verticalement en respectant la largeur respectivement la hauteur du conteneur.

BorderPane : C'est le BorderLayout swing, la fenêtre est décomposée en 5 zones top, bottom, right, left et center

StackPane : Les composants sont ajoutés l'un au-dessus de l'autre et il y a que le dernier qui est visible

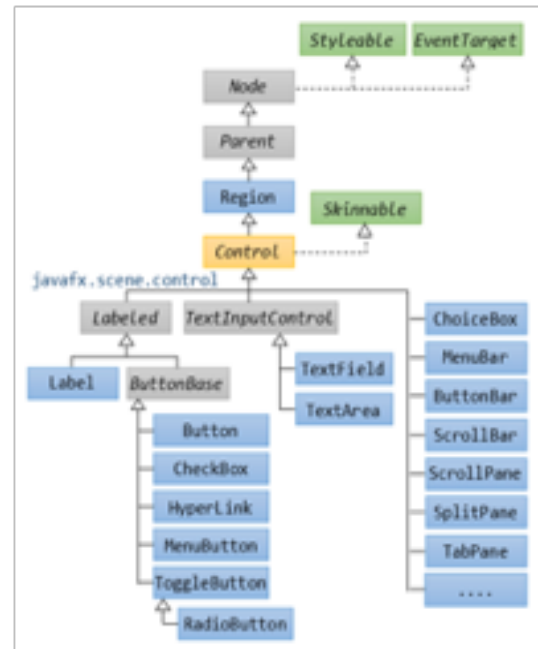
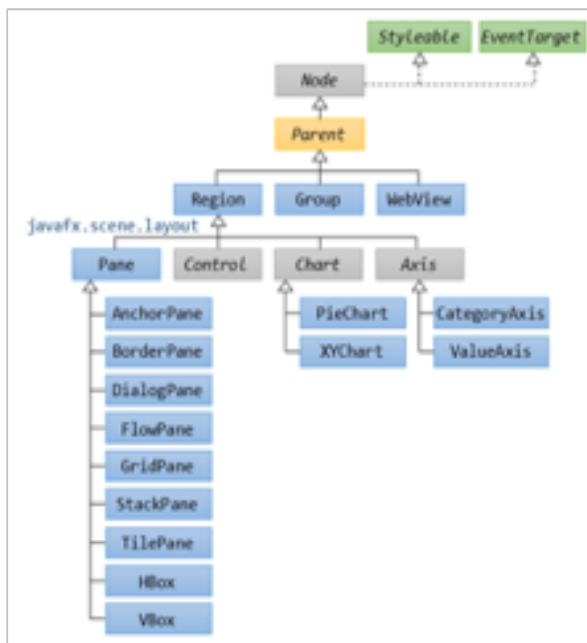
TilePane : équivalent au FlowPane il permet de placer tous les noeuds dans une grille dans laquelle chaque cellule a la même taille. Les nœuds peuvent être disposés horizontalement (en rangées) ou verticalement (en colonnes).

GridPane : C'est le GridLayout swing, Les composants sont ajustés selon une grille (lignes,colones)

AnchorPane : permet d'ancrer des nœuds en haut, en bas, à gauche, à droite ou au centre du volet. Lorsque la fenêtre est redimensionnée, les nœuds conservent leur position par rapport à leur point d'ancrage.

HBox : permet une disposition des composants en une seule ligne

VBox : permet une disposition des composants en une seule colonne



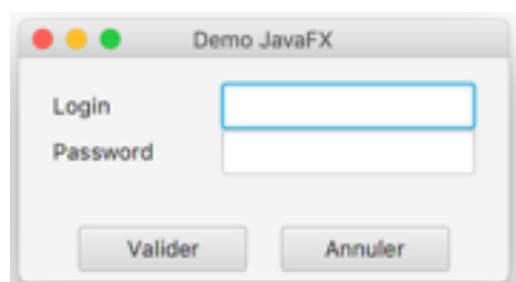
Les composants

- Accordion
- ChoiceBox
- ColorPicker
- ComboBox
- TableView
- TabPane
- TextArea
- TitledPane
- ToolBar
- TreeTableView
- TreeView
- DatePicker
- ListView
- Menu
- PasswordField
- ProgressBar
- Slider
- Spinner
- SplitMenuButton
- SplitPane



4. Exemple

La vue



```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.HBox?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane fx:controller= "application.AuthController"> ①
<center>

    <GridPane style="-fx-padding:15 20 15 20; " >
        <Label text="Login" GridPane.rowIndex="0" prefWidth="100" GridPane.columnIndex="0"/>
        <TextField prefWidth="150" GridPane.rowIndex="0" GridPane.columnIndex="1"/>
        <Label text="Password" GridPane.rowIndex="1" prefWidth="100"
        <TextField prefWidth="150" GridPane.rowIndex="1" GridPane.columnIndex="1"/>
```

```

GridPane.columnIndex="0"/>
    <TextField prefWidth="150" GridPane.rowIndex="1" GridPane.columnIndex="1"/>
</GridPane>
</center>
<bottom>
    <HBox style="-fx-padding:15 20 15 20; -fx-spacing:20px;" alignment="BASELINE_CENTER" >
        <Button text="Valider" prefWidth="100" onAction="#validateAction" /> ❷
        <Button text="Annuler" prefWidth="100" onAction="#cancelAction" />
    </HBox>
</bottom>
</BorderPane>

```

❶ L'attribut `fx:controller` permet de spécifier le nom de la classe contrôleur qui va par la suite traiter les événements

❷ Les méthodes `validateAction` et `cancelAction` représentent le traitement associé à l'événement Action (ici clique sur le bouton) seront implémenter dans le contrôleur

Le contrôleur

```

package application;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.Alert;
import javafx.scene.control.Button;
public class AuthController {

    // L'injection
    @FXML
    private void validateAction(ActionEvent event){}
    @FXML
    private void cancelAction(ActionEvent event) {}
}

```

5. ListView

```

<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.ListView?>
<?import javafx.scene.layout.VBox?>

<VBox xmlns="http://javafx.com/javafx/17" xmlns:fx="http://javafx.com/fxml/1">
    <children>
        <ListView fx:id="listView" />
    </children>
</VBox>

import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXML;
import javafx.scene.control.ListView;
public class ListViewController {
    @FXML
    private ListView<String> listView;
    public void initialize() {
        // Create a sample list of items
        ObservableList<String> items = FXCollections.observableArrayList(
            "Item 1", "Item 2", "Item 3", "Item 4", "Item 5");

        // Populate the ListView with the items
        listView.setItems(items);

        // Set an action event handler for item selection
        listView.setOnMouseClicked(event -> {
            String selectedItem =

```

```
listView.getSelectionModel().getSelectedItem();
        System.out.println("Selected Item: " + selectedItem);
    });
}
}
```

```
// Load the FXML file
FXMLLoader loader = new
FXMLLoader(getClass().getResource("ListViewExample.fxml"));
Parent root = loader.load();
// Set the controller for the FXML file
ListViewController controller = loader.getController();
```

6. Text Style

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.VBox?>

<VBox xmlns="http://javafx.com/javafx/17" xmlns:fx="http://javafx.com/fxml/
1" fx:controller="sample.Controller">
    <Label text="Styled Text" style="-fx-font-family: 'Arial'; -fx-font-
size: 16; -fx-font-weight: bold; -fx-fill: #336699; -fx-background-color:
#FFFFCC; -fx-padding: 10px; -fx-border-color: #336699; -fx-border-width:
2px; -fx-effect: dropshadow(three-pass-box, #666666, 10, 0, 0, 0);"/>
</VBox>
```

7. Charts

JavaFX est livré avec un ensemble de chartes graphiques prêtes à l'emploi :

AreaChart, BarChart, BubbleChart, LineChart, PieChart, ScatterChart, StackedAreaChart et StackedBarChart

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.BarChart;
import javafx.scene.chart.CategoryAxis;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.stage.Stage;

public class BarChartExample extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        // Create a BarChart
        BarChart<String, Number> barChart = createBarChart();
        Scene scene = new Scene(barChart, 600, 400);
        primaryStage.setTitle("JavaFX BarChart Example");

        primaryStage.setScene(scene);
        primaryStage.show();
    }

    private BarChart<String, Number> createBarChart() {
        // Define the axes
        CategoryAxis xAxis = new CategoryAxis();
        NumberAxis yAxis = new NumberAxis();
```

```

// Create the BarChart
BarChart<String, Number> barChart = new BarChart<>(xAxis, yAxis);
barChart.setTitle("BarChart Example");
// Create a data series
XYChart.Series<String, Number> series = new XYChart.Series<>();
series.setName("Data Series");

// Add data to the series
series.getData().add(new XYChart.Data<>("Category 1", 20));
series.getData().add(new XYChart.Data<>("Category 2", 50));
series.getData().add(new XYChart.Data<>("Category 3", 30));
series.getData().add(new XYChart.Data<>("Category 4", 40));
series.getData().add(new XYChart.Data<>("Category 5", 10));

// Add the series to the BarChart
barChart.getData().add(series);
return barChart;
}
public static void main(String[] args) {
    launch(args);
}
}

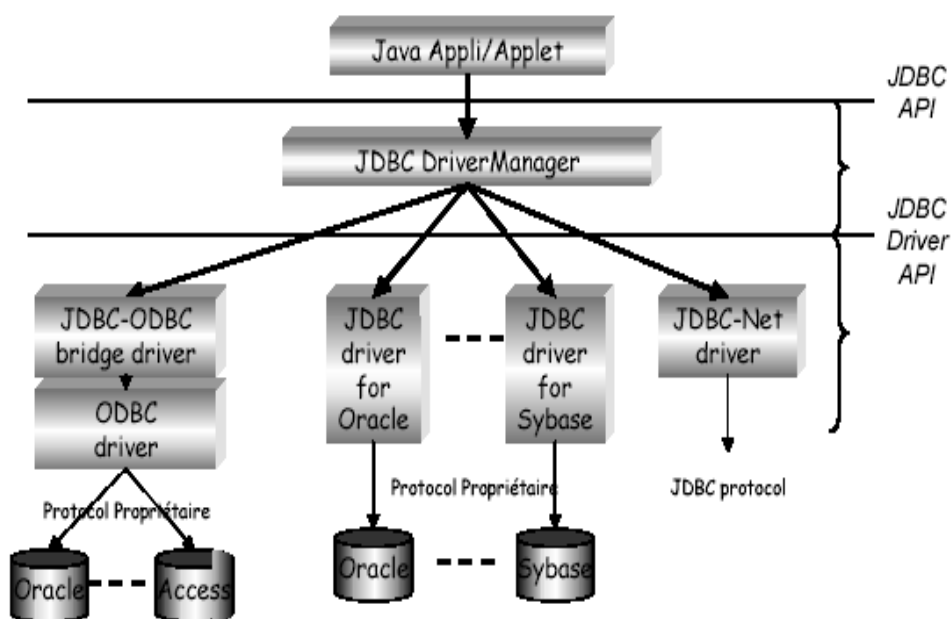
```

8 Accès aux bases de données en JDBC

1. Principe

JDBC est une API fournie avec Java permettant l'accès à n'importe quelle base de données locale ou à travers un réseau. Le principe de fonctionnement est comme suit :

- Chaque base de données utilise un pilote (*driver*) qui lui est propre et qui permet de convertir les requêtes JDBC dans le langage natif du SGBDR.
- Ces drivers dits JDBC (un ensemble de classes et d'interfaces Java) existent pour tous les principaux constructeurs : Oracle, Sybase, Informix, SQLServer, MySQL, MsAccess, ...



2. Mise en œuvre

Toutes les classes et les interfaces nécessaires à la connection à la base de données et son exploitation sont fournis par l'api java.sql. Les étapes à suivre pour mettre en place une application java basée sur JDBC sont :

1. Charger le driver JDBC

Utiliser la méthode de chargement à la demande **forName()** de la classe **Class** comme suit :

Class.forName("Nom du driver");

Exemples de drivers :

sun.jdbc.odbc.JdbcOdbcDriver : pour access fournis par JDK

oracle.jdbc.driver.OracleDriver : pour oracle non fournis

com.microsoft.jdbc.sqlserver.SQLServerDriver : SqlServer non fournis

Code de chargement :

```
try { Class.forName("com.mysql.jdbc.Driver").  
newInstance();
```

```
System.out.println("Tout est OK"); }
catch (Exception e) { System.out.println("Erreur de drivers
JDBC"); }
```

2. Établir la connexion à la base de données

La méthode **getConnection()** de **DriverManager** qui renvoie un objet de type **Connexion** permet d'ouvrir une connexion sur la base de données elle dispose de trois arguments : l'URL de la base de données, le nom de l'utilisateur de la base et son mot de passe.

La structure de l'URL doit respecter la forme <protocol>:<subprotocol>:<subname> , avec

protocol : jdbc

subprotocol : driver utilisé (mysql)

subname : //<host>[:<port>][/<databasename>]

Exemple : **jdbc:mysql://localhost:port (MySQL)/client**

Pour une base de donnée odbc **jdbc:odbc:NSD**, ou NSD est source de données associée à la base de données client exemple Access.

```
Connection cnx = DriverManager.getConnection(url,user,password) ;
```

Le **DriverManager** essaye tous les drivers qui se sont enregistrés.

3. Exécuter la requête

La première étape est de construire un objet **Statement**.

```
Statement stm = connection.createStatement() ;
```

Quand un objet **Statement** exécute une requête, il retourne un objet **ResultSet** (pour les requête de consultation). L'exécution de la requête se fait par l'une des trois méthodes suivantes :

- **executeQuery** : pour les requêtes de consultation. Renvoie un **ResultSet** pour récupérer les lignes une par une.
- **executeUpdate** : pour les requêtes de modification des données (update, insert, delete) ou autre requête SQL (create table, ...). Renvoie le nombre de lignes modifiées.
- **execute** : si on connaît pas à l'exécution la nature de l'ordre SQL à exécuter

```
ResultSet rsUsers = stm.executeQuery("SELECT * FROM T_Users;");
```

4. Traiter les données retournées

La manipulation des données se fait à travers un objet de type **ResultSet**. Différentes méthodes vous permettent de récupérer la valeur des champs de l'enregistrement courant.

Ces méthodes commencent toutes par **get**, immédiatement suivi du nom du type de données (ex: **getString**). Chaque méthode accepte soit l'indice de la colonne (à partir de 1) soit le nom de la colonne dans la table :

```
boolean getBoolean(int); boolean getBoolean(String); byte getByte(int); byte
getByte(String); Date getDate(int); Date getDate(String); double getDouble(int); double
getDouble(String); float getFloat(int); float getFloat(String); int getInt(int); int
```


getInt(String); long getLong(int); long getLong(String); short getShort(int); short getShort(String); String getString(int); String getString(String);

```
String strQuery = "SELECT * FROM T_Users;";
ResultSet rsUsers = stUsers.executeQuery(strQuery);
while(rsUsers.next()) {
    System.out.print("Id[" + rsUsers.getInt(1) + "]" + "Pass[" +
rsUsers.getString("Password") + "]); }

```

Pour parcourir un ResultSet dans tous les sens (premier, dernier, suivant et précédent) Il faut passer des paramètres supplémentaires à la méthode **createStatement** de l'objet de connexion : `stm = conn.createStatement(type, mode);`

- Type :
 - `ResultSet.TYPE_FORWARD_ONLY`
 - `ResultSet.TYPE_SCROLL_SENSITIVE`
 - `ResultSet.TYPE_SCROLL_INSENSITIVE`
- Mode :
 - `ResultSet.CONCUR_READ_ONLY`
 - `ResultSet.CONCUR_UPDATABLE`

Pour le type scroll, on a les possibilités de parcours first, last , next et previous. A titre d'exemple pour modifier le premier enregistrement de la table T_Users

```
rsUsers.first(); // Se positionne sur le premier enregistrement
rsUsers.updateString("Password", "toto"); // Modifie la valeur du password
rsUsers.updateRow(); //Applique les modifications sur la base

```

Avant de quitter l'application il faut fermer les différents espaces ouverts à savoir rsUsers, stm et cnx

3. PreparedStatement et CallableStatement

PreparedStatement : Requête dynamiques pré-compilée plus rapide qu'un **Statement** classique. Le SGBD n'analyse qu'une seule fois la requête pour de nombreuses exécutions d'une même requête SQL avec des paramètres variables. La méthode **prepareStatement()** throws **SQLException** de l'objet **Connection** crée un **PreparedStatement**

```
PreparedStatement ps =
con.prepareStatement("UPDATE emp SET sal = ? WHERE name = ?");
for(int i = 0; i < 10; i++) {
ps.setFloat(1, salary[i]);
ps.setString(2, name[i]);
int count = ps.executeUpdate(); }

```

Les arguments dynamiques sont spécifiés par un "?" ils sont ensuite positionnés par les méthodes **setInt()** , **setString()** , **setDate()** , ... de **PreparedStatement** et **setNull()** positionne le paramètre à NULL (SQL). Ces méthodes nécessitent 2 arguments, le premier (int) indique le numéro relatif de l'argument dans la requête et le second indique la valeur à positionner.

CallableStatement : utilisée pour exécuter une procédure sql stockée. Exemple d'addtion

```
/*
        --EXECUTE ADDITION 10,25,NULL
        ALTER PROCEDURE ADDITION
        @A INT
        , @B INT
        , @C INT OUT
        AS
        SELECT @C = @A + @B
    */
CallableStatement cs2 = con.prepareCall("{call ADDITION(?,?,?)}");
cs2.registerOutParameter(3, java.sql.Types.INTEGER);
cs2.setInt(1,10);
cs2.setInt(2,25);
cs2.execute();
int res = cs2.getInt(3);
System.out.println(res);
```

4. Les métadonnées

JDBC permet de récupérer des informations sur le type de données que l'on vient de récupérer par un SELECT (interface ResultSetMetaData), mais aussi sur la base de données elle-même (interface DatabaseMetaData). Les données que l'on peut récupérer avec DatabaseMetaData dépendent du SGBD avec lequel on travaille.

Récupération des méta-données select

```
ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
ResultSetMetaData rsmd = rs.getMetaData();
int nbColonnes = rsmd.getColumnCount();
for (int i = 1; i <= nbColonnes; i++) {
    String typeColonne = rsmd.getColumnTypeName(i);
    String nomColonne = rsmd.getColumnName(i);
    System.out.println("Colonne " + i + " de nom " + nomColonne +
        " de type " + typeColonne);}
```

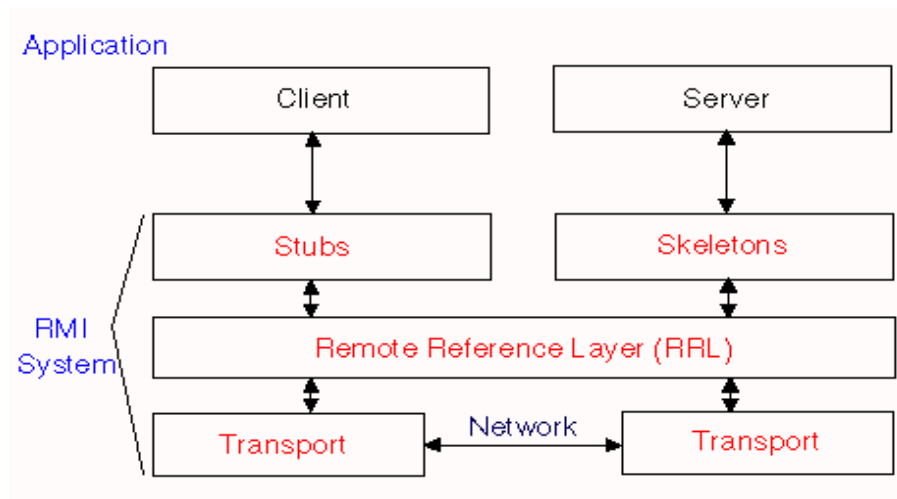
Récupération des méta-données de la base

```
private DatabaseMetaData metaData;
private Vector<String> listTables = new Vector();
. . .
metaData = conn.getMetaData();
// Récupérer les tables et les vues
String[] types = { "TABLE", "VIEW" };
ResultSet rs = metaData.getTables(null, null, "%", types);
while (rs.next()) {
    listTables.add(rs.getString(3));}
```

9. Client / Serveur en RMI

1. Définition

Appel de méthode à distance (Remote Methode Invocation) est une technologie sun développée depuis JDK 1.1 qui permet à un objet java s'exécutant sur une machine cliente d'invoquer des méthodes sur un objet s'exécutant sur une autre machine serveur. L'API rmi décharge le développeur de toute connaissance sur les réseaux (protocoles TCP, socket, format de données,...) et l'appelle de méthodes se fait d'une façon transparente.



2. Mise en œuvre

1. principe

Le développement d'une application distribuée RMI peut être décomposé en deux parties :

Côté serveur

- La définition d'une interface qui contient les méthodes qui peuvent être appelées à distance
- L'écriture d'une classe qui implémente cette interface
- L'enregistrement de l'objet distant dans le registre de noms RMI (RMI Registry)

Côté client

- L'obtention d'une référence sur l'objet distant à partir de son nom
- L'appel à la méthode à partir de cette référence

2. Exemple

L'objectif est de mettre en place une application rmi qui permettra à un client d'appeler une méthode addition(a,b) sur un objet distant calcule.

- Définition de l'interface calcul qui contiendra les services à appeler à distance :

```
import java.rmi.*;
public interface Calcul extends Remote {
    public double addition(double a, double b) throws RemoteException;}

```

- Implémentation de l'objet distant qui va implémenter l'interface Remote

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class CalculImpl extends UnicastRemoteObject implements Calcul{
    protected InformationImpl() throws RemoteException {
        super();}
    public double addition(double x, double y) throws RemoteException {
        return a+b ;}
}

```

- Enregistrement dans l'annuaire RmiRegistry (Main)

```
import java.rmi.registry.LocateRegistry;
import java.rmi.Naming;
class Main{
    public static void main(String[] args) {
        int port=1099;
        LocalRegistry.createRegistry(port)
        CalculImpl srv = new CalculImpl();
        String url = "rmi://@serveur:"+port+"/Reference";
        Naming.rebind(url, srv);
        System.out.println("Serveur lance :");
    } catch (Exception e) { e.printStackTrace();}
}
}

```

- Côté client en recherché l'objet distant par sa référence dans l'annuaire et appeler la méthode.

```
import java.rmi.Naming;
import java.rmi.Remote;
public class Client {
    public static void main(String[] args) {
        try {
            Remote serveur = (Calcul)Naming.lookup("rmi://@serveur:1099/Reference ");
            System.out.println("Resulta = " + serveur.addition(10,20));
        }
        catch (NotBoundException e) {
            e.printStackTrace();
        }
    }
}

```