- Java
  - Java
  - Java - Caractéristiques
  - Java development Kit
  - Java - Premier programme
  - Primitives du langage
    - Java - Types de données
    - Java - Variables
    - Java - Opérateurs
    - Java - Structures conditionnelles
  - Classes Java
    - Classes Java
    - Java - Méthodes
    - Java - Objets
    - Java - Constructeur
    - Java - Héritage
    - Java - Accessibilité
    - Java - Packages
    - Java - Exceptions
    - Java – JDBC

Java est un langage objet ressemblant au langage C++. Il a été mis au point en 1991 par la firme Sun Microsystems. Le but de Java à l'époque était de constituer un langage de programmation pouvant être intégré dans les appareils électroménagers, afin de pouvoir les contrôler, de les rendre interactifs, et surtout de permettre une communication entre les appareils. Ce programme de développement se situait dans un projet appelé *Green*, visant à créer une télécommande universelle (*Star 7*) comprenant un système d'exploitation capable de gérer l'ensemble des appareils électroménagers de la maison. Etant donné que le langage C++ comportait trop de difficultés, James Gosling, un des acteurs du projet (considéré désormais comme le père de Java) décida de créer un langage orienté objet reprenant les caractéristiques principales du C++, en éliminant ses points difficiles, et en le rendant moins encombrant et plus portable (il devait pouvoir être intégré dans n'importe quel appareil...). Ainsi, ce langage fut baptisé dans un premier temps *Oak* (*Oak* signifiant *chêne*). Toutefois, puisque ce nom était déjà utilisé, il fut rebaptisé *Java* en l'honneur de la boisson préférée des programmeurs, c'est-à-dire le café, dont une partie de la production provient de l'île Java.

A la même époque, le Web faisait son apparition, or *Java* possèdait toutes les caractéristiques faisant de lui un langage approprié pour le Web :

- Le réseau des réseaux rassemblant sur une même structure des machines différentes, il fallait un langage capable de fonctionner sur chacune d'entre-elles: Java était conçu pour être portable
- Le web était limité en bande passante: Java était conçu pour être petit

Ainsi, en 1994, l'équipe décida de mettre au point un navigateur (baptisé *HotJava*) intégrant Java et capable de faire fonctionner des *applets* (des petites applications fonctionnant dans un

navigateur). C'est ainsi que fin 1995 Java eut un terrible essor avec le soutien de Netscape, qui ne tarda pas à inclure Java dans son navigateur...

Comparaison de Java et de C++

Java est très proche du langage C++ étant donné qu'il a quasiment la même syntaxe. Toutefois Java est plus simple que le langage C++ bien qu'il s'en inspire, car les caractéristiques critiques du langage C++ (celles qui sont à l'origine des principales erreurs) ont été supprimées. Cela comprend :

- Les pointeurs
- La surcharge d'opérateurs
- L'héritage multiple
- La libération de mémoire est transparente pour l'utilisateur (il n'est plus nécessaire de créer de destructeurs)
- Une meilleure gestion des erreurs
- Les chaines et les tableaux sont désormais des objets faisant partie intégrante du langage

Toutefois Java est beaucoup moins rapide que le langage C++, il perd en rapidité ce qu'il gagne en portabilité...

Les versions de Java

Il existe trois versions majeures de Java :

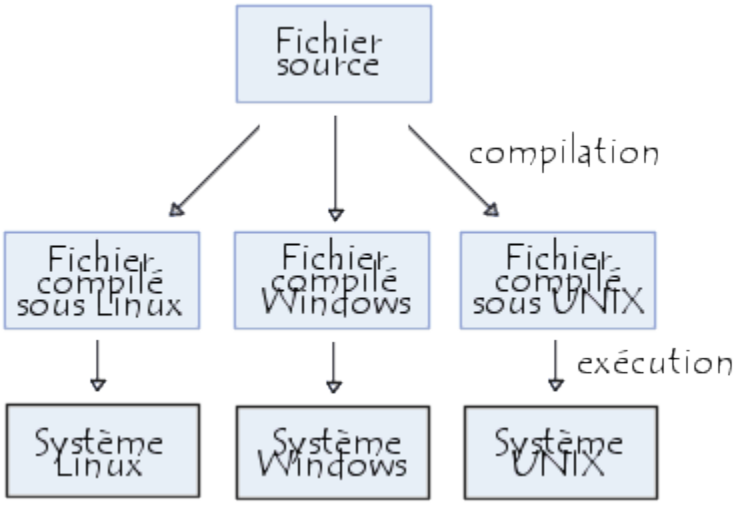
- Java 1.0: C'est la première version stable du langage, de ce fait la totalité des navigateurs la supporte
- Java 1.1: Cette version du langage a été publiée début 1997, elle apporte des améliorations syntaxiques ainsi que des progrès au niveau de l'interface utilisateur et de la gestion des exceptions
- Java 1.2: La version "2" apporte des améliorations multimédias. Elle permet par exemple l'utilisation d'interface utilisateur graphique avancée avec Swing, ainsi que la possibilité d'interagir avec elle par le moyen de glisser-déposer

Fichier source, compilation et machine virtuelle

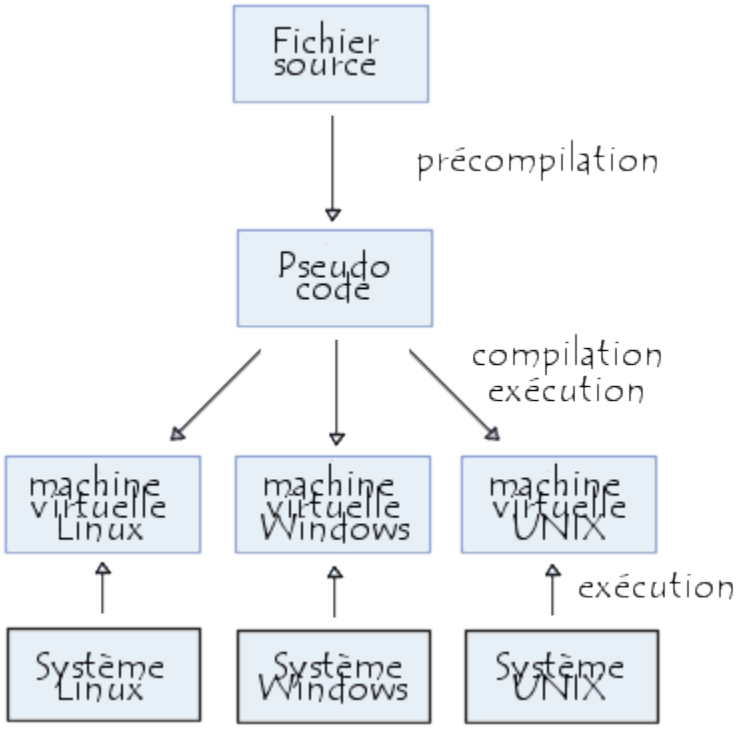
Le fichier source d'un programme écrit en Java est un simple fichier texte dont l'extension est par convention *.java*.
Ce fichier source doit être un fichier texte non formatté, c'est-à-dire un fichier texte dans sa plus simple expression, sans mise en forme particulière ou caractères spéciaux, c'est-à-dire qu'il contient uniquement les caractères ASCII de base.

Lorsque le programme est prêt à être "essayé", il s'agit de le compiler (le traduire en langage machine) à l'aide d'un compilateur. Toutefois, contrairement aux langages compilés traditionnels, pour lesquels le compilateur crée un fichier binaire directement exécutable par un processeur

donné (c'est-à-dire un fichier binaire contenant des instructions spécifiques à un processeur), le code source Java est compilé en un langage intermédiaire (appelé **pseudo-code** ou **bytecode**) dans un fichier portant le même nom que le fichier source à l'exception de son extension (*.class*).



Cette caractéristique est majeure, car c'est elle qui fait qu'un programme écrit en Java est portable, c'est-à-dire qu'il ne dépend pas d'une plate-forme donnée. En réalité le code intermédiaire n'est exécutable sur aucune plate-forme sans la présence d'une machine virtuelle, un interpréteur (la machine virtuelle est d'ailleurs parfois appelée *interpréteur Java*) tournant sur une plate-forme donnée, et capable d'interpréter le code intermédiaire.



Ainsi, pour peu qu'une plate-forme (windows 95, Unix, Linux, Amiga, ...) possède une machine virtuelle fonctionnant sous son système, celle-ci est capable d'exécuter n'importe quelle application Java!

De cette façon, l'ensemble des navigateurs permettant d'utiliser des applets possèdent une machine virtuelle...

Typologie

La manière d'écrire les choses en Java est importante. Le langage Java est par exemple sensible à la casse (en anglais *case sensitive*), entendez par là qu'un nom contenant des majuscules est différencié du même nom écrit en minuscules. Ainsi, les spécifications du langage Java précisent que la fonction principale doit être appelée *main()* et non *Main()* ou *MAIN()*.
D'autre part, toute instruction se termine par un point-virgule.

Applications et Applets

Java permet de créer deux types de programmes :

- **Les applications**: programmes tels qu'on les connaît, c'est-à-dire s'exécutant dans le système d'exploitation à condition d'avoir installé une machine virtuelle. Voici ce à quoi ressemble une application Java :

```
public class Nom_du_programme {
   public static void main (String args[]){
         System.out.println("Hello World");

   }
}
```

- **Les applets** (prononcez *Applettes*, et traduisez *Appliquettes*, le nom grotesque donné en français) : Il s'agit de petites applications destinées à fonctionner sur un navigateur. Ainsi une applet a un champ d'action beaucoup plus réduit qu'une application pour des raisons de sécurité (une applet ne peut par exemple pas accéder au système sur lequel elle s'exécute...). Voici un exemple d'applet :

```
public class Nom_de_l_applet extends java.applet.Applet {
   public void paint (Graphics g){
  g.drawString("Hello World",20,40);

   }
}
```

Ajout de commentaires

Lorsqu'un programme devient long et compliqué il peut être intéressant (il est même conseillé) d'ajouter des lignes de commentaires dans le programme, c'est-à-dire des portions du fichier

source qui ont pour but d'expliquer le fonctionnement du programme sans que le compilateur ne les prennent en compte (car il génèrerait une erreur).

Pour ce faire, il est possible d'utiliser des balises qui vont permettre de délimiter les explications afin que le compilateur les ignore et passe directement à la suite du fichier.
Ces délimiteurs sont /* et */. Un commentaire sera donc noté de la façon suivante :

```
/* Voici un commentaire! */
```

Il y a toutefois quelques règles à respecter :

- Les commentaires peuvent être placés n'importe où dans le fichier source
- Les commentaires ne peuvent contenir le délimiteur de fin de commentaire (*/)
- Les commentaires ne peuvent être imbriqués
- Les commentaires peuvent être écrits sur plusieurs lignes
- Les commentaires ne peuvent pas couper un mot du programme en deux

Le langage permet aussi de mettre toute la fin d'une ligne en commentaire en utilisant le double *slash* (*//*). Tout ce qui se situe à droite de ce symbole sera mis en commentaire.

Enfin, un dernier type de commentaires permet de documenter une application, de telle façon que le programme *JavaDoc* (fourni avec le JDK, *Java Development Kit*, le kit permettant de compiler du Java), puisse créer des documents HTML à partir de l'application afin de la documenter. Ces commentaires, vus comme des commentaires normaux par Java, sont repérés par les délimiteurs /** et */.

Qu'est-ce que le Java Development Kit

Le Java Development Kit, communément appelé *JDK*, est le kit de développement de base que propose gratuitement la firme *Sun Microsystem*. Le Kit de développement comprend plusieurs outils, parmi lesquels :

- **javac**: le compilateur Java
- **java**: un interpréteur d'applications (machine virtuelle)
- **applet viewer**: un interpréteur d'applets
- **jdb**: un débogueur
- **javap**: un décompilateur, pour revenir du bytecode au code source
- **javadoc**: un générateur de documentation
- **jar**: un compresseur de classes Java

Les versions

Il existe plusieurs versions du JDK sur chaque plateforme (Unix, Solaris, Windows 9x,...) :

- Les versions 1.0.x sont compatibles avec Java 1.0. Etant donné que la quasi-totalité des navigateurs supportent Java 1.0 il est conseillé d'utiliser la version 1.0.x la plus récente pour développer des applets
- Les versions 1.1.x sont compatibles avec Java 1.1
- Les versions 1.2.x sont compatibles avec Java 2

Il existe des versions plus récentes de Java (et donc du JDK), compatibles avec les informations présentes dans ce dossier.

Où se procurer le JDK

Le JDK est disponible gratuitement en téléchargement sur le site de *Sun Microsystems* :

- http://java.sun.com
- http://javasoft.com

Etant donné la taille du JDK (20 Mo), et de sa documentation (12 Mo), le téléchargement peut durer plusieurs heures. Ainsi, vous pouvez aussi trouver le JDK dans la plupart des livres traitant de Java, ou tout simplement dans un magazine en rapport avec la programmation (jetez un oeil du côté de votre libraire!).

Le compilateur

javac est un compilateur, c'est-à-dire qu'il transforme le code source en bytecode, un fichier binaire intermédiaire interprétable par la machine virtuelle sur n'importe quelle plate-forme.

javac s'utilise avec la syntaxe suivante :

```
javac -g nom_du_fichier.java
```

L'option -g permet tout simplement d'inclure dans le pseudo-code des informations de débogage afin de pouvoir utiliser le débogueur *jdb*.

L'interpréteur

L'interpréteur *java* est une machine virtuelle fonctionnant en mode texte, c'est-à-dire sans interface graphique. Sa syntaxe est la suivante :

```
java nom_du_fichier
```

L'interpréteur d'applets

appletviewer a pour but de pouvoir visualiser l'exécution d'un applet (il est aussi possible de la visualiser sur un navigateur compatible Java, comme Internet Explorer 4 ou supérieur, Netscape Navigator 4 ou supérieur, HotJava...). Sa syntaxe est la suivante :

```
appletviewer nom_de_l_applet.class
```

Javadoc

JavaDoc est un utilitaire permettant de créer une documentation au format HTML à partir de vos programmes, grâce aux commentaires prévus à cet effet (/**) que vous aurez incorporé dans le code.

Le compresseur de classes

Jar est un utilitaire permettant de compresser vos classes Java afin de réduire leur taille et de rendre leur téléchargement plus rapide.

Première application avec Java

La première chose à faire est de créer un simple fichier texte (sans mise en forme) et de taper les quelques lignes suivantes :

```
// Votre premiere application en Java
class FirstApp {
   public static void main (String[] args){
        System.out.println("Hello World");

   }
}
```

Etant donné que l'on a donné le nom de *FirstApp* à notre programme, il est essentiel de sauver ce fichier sous le nom *FirstApp.java* (un autre nom provoquera une erreur lors de la compilation). A l'aide du JDK (ou d'un environnement de développement Java intégré) compilez le programme en vous plaçant, sous une fenêtre DOS (en supposant que vous êtes sous un environnement *Microsoft*), grâce à la commande suivante :

```
javac FirstApp.java
```

Si tout se déroule normalement, le compilateur devrait créer un fichier nommé *FirstApp.class*. dans le cas contraire, vous avez soit une erreur dans votre programme (vérifiez la syntaxe) ou bien vous avez mal installé le JDK...

En supposant que le JDK a été installé correctement et que la syntaxe de votre programme est bonne, vous pouvez désormais exécuter votre programme grâce à la machine virtuelle Java, en tapant la commande suivante :

```
java FirstApp
```

```
Il ne faut pas spécifier l'extension (.class) au risque
d'obtenir une erreur !
```

```
Pour passer des arguments lors de l'exécution, il suffit de
préciser ces arguments après le nom du fichier :

java NomFichier arg1 arg2 ...
Si il y a plusieurs arguments, ceux-ci doivent être séparés par
au moins un espace.
```

Le programme devrait en toute logique afficher le message "Hello World" à l'écran. Cela est bien peu, mais constitue un bon début pour analyser le programme. Voyons ligne par ligne comment est constitué cette application :

- La première ligne

  ```
  // Votre premiere application en Java
  ```

  est une ligne de commentaire permettant d'ajouter des remarques sur l'application

- La seconde ligne

  ```
  class FirstApp {
  ```

  définit la classe principale de votre application grâce au mot clé *class*. Le nom que vous donnez à cette classe représente le nom de votre programme, cela signifie qu'une application Java possède au moins une classe, et que celle-ci possède un nom correspondant au nom du programme. Puisque Java est sensible à la casse, le nom que vous donnez au programme doit être rigoureusement le même que celui que vous donnez au fichier portant l'extension *.java*
  La seconde ligne se termine par une accolade ouvrante "{", correspondant au début d'un *bloc*, c'est-à-dire un ensemble de lignes de codes correspondant à la classe et délimité par deux accolades

- La troisième ligne

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

  définit la méthode principale de l'application, c'est-à-dire une procédure propre à la classe est exécutée lors de son appel. Le mot clé *void* précédant la méthode *main()* indique le type de retour de la fonction, c'est-à-dire le type de valeur que la méthode doit retourner. Dans le cas présent, il s'agit du type *void* (signifiant *rien* en anglais), donc la méthode ne doit retourner aucune valeur. Les parenthèses suivant le mot *main* permettent de définir les paramètres de la méthode, c'est-à-dire les données qu'elle reçoit en entrée. Dans cette application (et dans toutes vos futures applications) le paramètre est *String args[]*, ce qui correspond à un ensemble (tableau) de chaînes de caractères. Cela correspond en réalité aux paramètres que vous placez après la ligne *javac FirstApp.class*, si, par exemple, vous voulez pouvoir préciser des options lors de l'exécution de votre application. Vous rencontrerez parfois l'écriture *String[] args*, cette écriture est juste elle aussi, voire plus logique... (cela sera expliqué plus loin).

Enfin les mots clés *public* et *static* définissent les caractéristiques de la méthode *main()*. Le mot clé *public* définit les classes pouvant exécuter la méthode *main()*, *public* signifie donc que toutes les classes peuvent y avoir accès. Or, l'interpréteur Java étant "considéré comme une classe", il est essentiel qu'il puisse y avoir accès afin de lancer l'application... Comme dans le cas de la classe *FirstApp*, la ligne déclarant la méthode *main()* se termine par une accolade ouvrante indiquant un bloc correspondant au corps de la méthode *main()*

- La quatrième ligne

```
System.out.println("Hello World");
```

  correspond au corps de la méthode *main()*. Il s'agit de l'appel de la méthode *println()*, possèdant un paramètre (une chaîne de caractère). Cette méthode permet d'afficher sur la sortie standard (ici, l'écran) la chaîne de caractère qui lui est passée en paramètre, en lui ajoutant automatiquement un caractère de retour à la ligne (il existe aussi une méthode appelée *print()*, n'ajoutant pas de retour à la ligne). Cette méthode est précédée de *System.out.*, cela signifie qu'elle fait partie de l'objet *out*, faisant lui-même partie de la classe *System*. En réalité toutes les méthodes natives de Java font partie d'objets, répartis dans une hiérarchie de classes. Ainsi, la classe *System* fait partie de la classe *lang*, qui fait elle-même partie de la classe *java*. Toutefois, il n'est pas nécessaire de préciser *java.lang.System.out*, car la classe *java.lang* est la seule classe directement accessible (celle-ci contient en effet les méthodes essentielles). De cette façon, la ligne "*System.out.println("Hello World");*" est équivalente à "*java.lang.System.out.println("Hello World");*".

- Une application Java possède au moins une classe
- Le nom de la classe correspond au nom de l'application (avec la même casse, c'est-à-dire orthographié de la même façon aux majuscules près)
- Une application Java possède une méthode *main()* déclarée en public et en static
- Le paramètre de la méthode *main()* peut s'écrire indifféremment *String args[]* ou *String[] args*

Première applet avec Java

Les applets sont différentes des applications dans la mesure où il s'agit de petites applications graphiques destinées à s'exécuter dans un navigateur internet. D'autre part, les applets ont un champ d'action limité pour des raisons de sécurité puisqu'elles s'exécutent dans un environnement distant. Elles ne peuvent donc pas accéder à toutes les ressources de l'environnement distant (lecture/écriture sur le disque par exemple).

Voici donc le code d'une applet toute simple :

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

```
public class FirstApplet extends java.applet.Applet {
   public void init (){
        add(new Label("Hello World"));

   }
}
```
Le programme devrait en toute logique afficher le message "Hello World" à l'écran sur le navigateur, à condition de créer le fichier HTML approprié "hébergeant" l'applet :
```
<HTML>

<BODY>

<APPLET code="FirstApplet.class" width="100" height="30" align="left">

Message s'affichant si l'applet n'est pas supportée par le navigateur
</APPLET>

</BODY>

</HTML>
```
Pour visualiser l'applet, il vous suffit donc d'enregistrer ce fichier avec l'extension *.htm* ou *.html*, comme par exemple *FirstApplet.htm*.

Pour l'exécuter vous pouvez :

- soit utiliser *appletviewer*, en vous plaçant dans le répertoire contenant l'applet et le fichier HTML, et taper la commande :

  ```
  appletviewer FirstApplet.htm
  ```

- soit utiliser votre navigateur en ouvrant le fichier HTML

L'exécution de cette applet devrait donner un résultat comme celui-ci :

Voyons ligne par ligne comment est constitué cette applet :

- La première ligne

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

  indique au compilateur qu'il a besoin de l'ensemble des classes situées dans le package *awt* pour pouvoir être compilé correctement. Les classes AWT, représente une bibliothèque de méthodes graphiques permettant d'afficher des éléments graphiques sur le navigateur

- La seconde ligne

```
public class FirstApplet extends java.applet.Applet {
```

définit la classe principale de votre applet grâce au mot clé *class*. Le nom que vous donnez à cette classe représente le nom de votre programme, cela signifie qu'une applet Java possède au moins une classe, et que celle-ci possède un nom correspondant au nom du programme. Le nom que vous donnez au programme doit,comme pour les applications, être rigoureusement le même que celui que vous donnez au fichier portant l'extension *.java*
Le nom de la classe est précédée du mot clé *public*, permettant l'utilisation de cette classe par le navigateur. Elle est aussi suivie de la mention *extends java.applet.Applet*, indiquant que la classe que l'on définit hérite d'une classe existante appelée *Applet*. Cette classe définit une méthode *init()* vide, ce qui signifie que l'applet posséde par défaut une méthode *init()*, qui peut être redéfinie
La seconde ligne se termine par une accolade ouvrante "{", correspondant au début d'un *bloc*, c'est-à-dire un ensemble de lignes de codes correspondant à la classe et délimité par deux accolades

- La troisième ligne

```
   public void init (){
```

redéfinit la méthode *init()* de l'applet. Celle-ci ne possède pas d'argument et ne retourne aucune valeur (son type de retour est *void*). Par contre cette méthode doit obligatoirement être définie en tant que *public* pour pouvoir être exécutée correctement

- La quatrième ligne

```
  add(new Label("Hello World");
```

correspond au corps de la méthode *init()*. Il s'agit de l'appel de la méthode *add()*, possèdant un paramètre (un Label, objet de la classe *java.awt* permettant d'afficher la chaîne de caractère passée en paramètre). La méthode *add()* fait en fait partie de la classe *java.awt.Panel*, qui est une classe parente de *java.applet.Applet*. De cette façon, *java.applet.Applet* possède l'ensemble des méthodes qui sont définies en son sein, ainsi que l'ensemble des méthodes définies dans les classes dont elle hérite...

Voici donc deux exemples d'application et d'applet permettant d'avoir un aperçu de ce à quoi peut ressembler un programme Java (qu'il soit une application ou une applet). La section suivante présente les primitives du langage Java, c'est-à-dire les briques élémentaires pouvant être utilisées dans une classe...

Les primitives

Java est un langage orienté objet, c'est-à-dire que les éléments manipulés sont des classes, ou plus exactement des objets, c'est-à-dire des instances de classes.

Toutefois ces objets contiennent des données possèdant un type (et une représentation). Ces données sont un ensemble d'éléments stockés en mémoire et baptisés pour l'occasion **primitives**. Les données manipulées avec Java, sont typées, c'est-à-dire que pour chaque donnée que l'on utilise (dans les variables par exemple) il faut préciser le type de donnée, ce qui permet de connaître l'occupation mémoire (le nombre d'octets) de la donnée ainsi que sa représentation, cela peut être entre autres :

- **des nombres entiers**: (**int**)
- **réels**, c'est-à-dire à virgules (**float**)

De plus, le langage Java introduit un type de donnée appelée *boolean*. Ce type de variable accepte deux états :

- **True** (vrai): correspondant à une valeur *vraie*
- **False** (faux): correspondant à une valeur *fausse*

En réalité, toutes les valeurs différentes de *True* sont considérées comme fausses (donc considérées comme égales à *False*). Si la variable n'est pas initialisée, sa valeur par défaut est *False*.

Voici un tableau répertoriant les primitives (types de données) de Java :

Primitive	Signification	Taille (en octets)	Plage de valeurs acceptée
char	Caractère	2	valeur du jeu de caractères Unicode (65000 caractères possibles)
byte	Entier très court	1	-128 à 127
short	Entier court	2	-32768 à 32767
int	Entier	4	-2 147 483 648 à 2 147 483 647
long	Entier long	8	-9223372036854775808 à 9223372036854775807
float	flottant (réel)	4	$-1.4*10^{-45}$ à $3.4*10^{38}$
double	flottant double	8	$4.9*10^{-324}$ à $1.7*10^{308}$
boolean	booléen	1	0 ou 1 (en réalité, toute autre valeur

que 0 est considérée égale à 1)

Les enveloppeurs (Wrappers)

Les primitives peuvent être "enveloppées" dans un objet provenant d'une classe prévue à cet effet et appelée *Wrapper* (mot anglais signifiant *enveloppeur*). Les enveloppeurs sont donc des objets pouvant contenir une primitive et auxquels sont associés des méthodes permettant de les manipuler.

Voici la liste des enveloppeurs disponibles avec Java :

Enveloppeur	**Primitive associée**
BigDecimal	aucune primitive associée, car cette classe peut contenir des valeurs décimales de précision quelconque
BigInteger	aucune primitive associée, car cette classe peut contenir des valeurs entières de précision quelconque
Character	char
Byte	byte
Short	short
Integer	int
Long	long
Float	float
Double	double
Boolean	boolean
Void	void

Nombre entier (*int*)

Un nombre entier est un nombre sans virgule qui peut être exprimé dans différentes bases :

- Base décimale: L'entier est représenté par une suite de chiffre unitaires (de 0 à 9) ne devant pas commencer par le chiffre 0

- Base hexadécimale: L'entier est représenté par une suite d'unités (de 0 à 9 ou de A à F (ou a à f)) devant commencer par *0x* ou *0X*
- Base octale: L'entier est représenté par une suite d'unités (incluant uniquement des chiffres de 0 à 7) devant commencer par *0*

Les entiers sont signés par défaut, cela signifie qu'ils comportent un signe. Pour stocker l'information concernant le signe (en binaire), les ordinateurs utilisent le complément à deux

Nombre à virgule (*float*)

Un nombre à virgule flottante est un nombre à virgule, il peut toutefois être représenté de différentes façons :

- un entier décimal: 895
- un nombre comportant un point (et non une virgule): 845.32
- une fraction: 27/11
- un nombre exponentiel, c'est-à-dire un nombre (éventuellement à virgule) suivi de la lettre *e* (ou *E*), puis d'un entier correspondant à la puissance de 10 (signé ou non, c'est-à-dire précédé d'un + ou d'un -)
  2.75e-2
  35.8E+10
  .25e-2

En réalité, les nombres réels sont des nombres à virgule flottante, c'est-à-dire un nombre dans lequel la position de la virgule n'est pas fixe, et est repérée par une partie de ses bits (appelée l'exposant), le reste des bits permettent de coder le nombre sans virgule (la mantisse).

Les nombres de type **float** sont codés sur 32 bits dont :

- 23 bits pour la mantisse
- 8 bits pour l'exposant
- 1 bit pour le signe

Les nombres de type **double** sont codés sur 64 bits dont :

- 52 bits pour la mantisse
- 11 bits pour l'exposant
- 1 bit pour le signe

La précision des nombres réels est approchée. Elle dépend par le nombre de positions décimales, suivant le type de réel elle sera au moins :

- de 6 chiffres après la virgule pour le type **float**
- de 15 chiffres après la virgule pour le type **double**

Caractère (*char*)

Le type **char** (provenant de l'anglais *character*) permet de stocker la valeur Unicode, codée sur 16 bits, d'un caractère, c'est-à-dire un nombre entier codé sur 16 bits, soit 65000 caractères ! Par conséquent il est possible de stocker un caractère accentué dans une variable de type *char*.

Si jamais on désire par exemple stocker la lettre B, on pourra définir cette donnée soit par son code Unicode (\u0066), soit en notant 'B' ou les apostrophes simples signifient *code ascii de...*

Chaînes de caractère (*String*)

Les chaînes de caractères ne correspondent pas à un type de données mais à une classe, ce qui signifie qu'une chaîne de caractère est un objet possèdant des attributs et des méthodes. Une chaîne peut donc être déclarée de la façon suivante :

```
String s = "Chaine de caractères";
```

Conversion de type de données (casting)

On appelle *conversion de type de données*, parfois *transtypage* (traduction de l'anglais *casting*), le fait de modifier le type d'une donnée en une autre. Il peut arriver par exemple que l'on veuille travailler sur un type de variable, puis l'utiliser sous un autre type. Imaginons que l'on travaille par exemple sur une variable en virgule flottante (type *float*), il se peut que l'on veuille "supprimer les chiffres après la virgule", c'est-à-dire convertir un *float* en *int*. Cette opération peut être réalisée de deux manières :

- **conversion implicite**: une conversion implicite consiste en une modification du type de donnée effectuée automatiquement par le compilateur. Cela signifie que lorsque l'on va stocker un type de donnée dans une variable déclarée avec un autre type, le compilateur ne retournera pas d'erreur mais effectuera une conversion *implicite* de la donnée avant de l'affecter à la variable.
- `int x;`
- ```
  x = 8.324;
  ```

  x contiendra après affectation la valeur 8

- **conversion explicite**: une conversion explicite (appelée aussi *opération de cast*) consiste en une modification du type de donnée forcée. Cela signifie que l'on utilise un opérateur dit *de cast* pour spécifier la conversion. L'opérateur de cast est tout simplement le type de donnée, dans lequel on désire convertir une variable, entre des parenthèses précédant la variable.
- `int x;`
- ```
  x = (int)8.324;
  ```

  x contiendra après affectation la valeur 8

De plus, le langage Java rajoute une notation fonctionnelle pour faire une conversion explicite, la syntaxe de cette notation est :

```
int x;

x = int(8.324);
```

Le concept de variable

Une variable est un objet repéré par son nom, pouvant contenir des données, qui pourront être modifiées lors de l'exécution du programme. Les variables en langage Java sont typées, c'est-à-dire que les données contenues dans celles-ci possèdent un type, ainsi elles sont donc stockées à une adresse mémoire et occupent un nombre d'octets dépendant du type de donnée stockée.

Avec Java, les noms de variables peuvent être aussi long que l'on désire, toutefois le compilateur ne tiendra compte "que" des 247 premiers caractères. De plus, elles doivent répondre à certains critères :

- un nom de variable doit commencer par une lettre (majuscule ou minuscule) ou un "_" (pas par un chiffre)
- un nom de variables peut comporter des lettres, des chiffres et le caractère _ (les espaces ne sont pas autorisés!)
- Les noms de variables ne peuvent pas être les noms suivants (qui sont des noms réservés) :
    - abstract
    - boolean break byte byvalue
    - case cast catch char class const continue
    - default do double
    - else extends
    - false final finally float for future
    - generic goto
    - if implements import inner instanceof int interface
    - long
    - native new null
    - operator outer
    - package private protected public
    - return rest return
    - short static sctrictfp super switch synchronized
    - this throw throws transient true try
    - var void volatile
    - while widefp

Nom de variable correct	**Nom de variable incorrect**	**Raison**
Variable	Nom de Variable	comporte des espaces

Nom_De_Variable	123Nom_De_Variable	commence par un chiffre
nom_de_variable	toto@mailcity.com	caractère spécial @
nom_de_variable_123	Nom-de-variable	signe - interdit
_707	this	nom réservé

Les noms de variables sont sensibles à la casse (Java fait la différence entre un nom en majuscule et un nom en minuscules), il faut donc veiller à utiliser des noms comportant la même casse!

La déclaration de variables

Pour pouvoir utiliser une variable, il faut la définir, c'est-à-dire lui donner un nom, mais surtout un type de donnée à stocker afin qu'un espace mémoire conforme au type de donnée qu'elle contient lui soit réservé.

Une variable se déclare de la façon suivante :

```
type Nom_de_la_variable;
```
ou bien s'il y a plusieurs variables du même type :
```
type Nom_de_la_variable1, Nom_de_la_variable2, ...;
```

- Java impose que les variables soient impérativement déclarées
- Java permet de définir une variable à n'importe quel endroit du code, afin de le rendre plus lisible

Affectation d'une donnée à une variable

Pour stocker une donnée dans une variable que l'on a initialisée, il faut faire une affectation, c'est-à-dire préciser la donnée qui va être stockée à l'emplacement mémoire qui a été réservé lors de l'initialisation.

Pour cela on utilise l'opérateur d'affectation "=" :

```
Nom_de_la_variable = donnée;
```

Pour stocker le caractère B dans la variable que l'on a appelée *Caractere*, il faudra écrire :

```
Caractere = 'B';
```

Ce qui signifie *stocker la valeur ASCII de "B" dans la variable nommée "Caractere"*. Il est bien évident qu'il faut avoir préalablement déclaré la variable en lui affectant le type *char* :
```
 char Caractere;
```

Initialisation d'une variable

La déclaration d'une variable ne fait que "réserver" un emplacement mémoire où stocker la variable. Tant que l'on ne lui a pas affecté une donnée celle-ci contient ce qui se trouvait précédemment à cet emplacement, que l'on appelle *garbage* (en français: *détritus*).

On peut donc affecter une valeur initiale à la variable lors de sa déclaration, on parle alors d'initialisation :

```
type Nom_de_la_variable = donnee;
```

Par exemple :

```
float toto = 125.36f;
```

Portée (visibilité) des variables

Selon l'endroit où on déclare une variable, celle-ci pourra être accessible (visible) de partout dans le code ou bien que dans une portion confinée de celui-ci (à l'intérieur d'une fonction par exemple), on parle de *portée* (ou *visibilité*) d'une variable.

Lorsqu'une variable est déclarée dans le code même, c'est-à-dire à l'extérieur de toute fonction ou de tout bloc d'instruction, elle est accessible de partout dans le code (n'importe quelle fonction du programme peut faire appel à cette variable). On parle alors de **variable globale**

Lorsque l'on déclare une variable à l'intérieur d'un bloc d'instructions (entre des accolades), sa portée se confine à l'intérieur du bloc dans lequel elle est déclarée.

- Une variable déclarée au début du code, c'est-à-dire avant tout bloc de donnée, sera globale, on pourra alors les utiliser à partir de n'importe quel bloc d'instruction
- Une variable déclarée à l'intérieur d'un bloc d'instructions (dans une fonction ou une boucle par exemple) aura une portée limitée à ce seul bloc d'instruction, c'est-à-dire qu'elle est inutilisable ailleurs, on parle alors de variable locale

D'une manière générale il est tout de même préférable de donner des noms différents aux variables locales et globales...

Définition de constantes

Une constante est une variable dont la valeur est inchangeable lors de l'exécution d'un programme. Java ne possède pas véritablement de constantes. Toutefois, le mot clé *final* permet de définir une variable dont la valeur ne peut pas être modifiée après son initialisation.

```
final int Variable = 12;
```

aura pour effet de définir une variable de type *entier* possèdant la valeur 12 et ne pouvant pas être modifiée dans la suite du code, auquel cas le compilateur génère une erreur...

Il s'agit là d'une utilisation restrictive du mot clé *final*, car celui-ci permet d'affecter une valeur différente à chaque exécution du programme, tout en garantissant que la variable, une fois initialisée avec une valeur, ne subira plus aucune modification.

Qu'est-ce qu'un opérateur?

Les opérateurs sont des symboles qui permettent de manipuler des variables, c'est-à-dire effectuer des opérations, les évaluer, ...
On distingue plusieurs types d'opérateurs :

- les opérateurs de calcul
- les opérateurs d'assignation
- les opérateurs d'incrémentation
- les opérateurs de comparaison
- les opérateurs logiques
- (les opérateurs bit-à-bit)
- (les opérateurs de rotation de bit)

Les opérateurs de calcul

Les opérateurs de calcul permettent de modifier mathématiquement la valeur d'une variable

Opérateur	Dénomination	Effet	Exemple	Résultat (avec x valant 7)
+	opérateur d'addition	Ajoute deux valeurs	x+3	10
-	opérateur de soustraction	Soustrait deux valeurs	x-3	4
*	opérateur de multiplication	Multiplie deux valeurs	x*3	21
/	plus: opérateur de division	Divise deux valeurs	x/3	2.3333333
=	opérateur d'affectation	Affecte une valeur à une variable	x=3	Met la valeur 3 dans la variable x

Les opérateurs d'assignation

Ces opérateurs permettent de simplifier des opérations telles que *ajouter une valeur dans une variable et stocker le résultat dans la variable*. Une telle opération s'écrirait habituellement de la façon suivante par exemple: *x=x+2*
Avec les opérateurs d'assignation il est possible d'écrire cette opération sous la forme suivante: *x+=2*
Ainsi, si la valeur de x était 7 avant opération, elle sera de 9 après...

Les autres opérateurs du même type sont les suivants :

Opérateur	Effet
+=	addition deux valeurs et stocke le résultat dans la variable (à gauche)
-=	soustrait deux valeurs et stocke le résultat dans la variable
*=	multiplie deux valeurs et stocke le résultat dans la variable
/=	divise deux valeurs et stocke le résultat dans la variable

Les opérateurs d'incrémentation

Ce type d'opérateur permet de facilement augmenter ou diminuer d'une unité une variable. Ces opérateurs sont très utiles pour des structures telles que des boucles, qui ont besoin d'un compteur (variable qui augmente de un en un).

Un opérateur de type *x++* permet de remplacer des notations lourdes telles que *x=x+1* ou bien *x+=1*

Opérateur	Dénomination	Effet	Syntaxe	Résultat (avec x valant 7)
++	Incrémentation	Augmente d'une unité la variable	x++	8
--	Décrémentation	Diminue d'une unité la variable	x--	6

Les opérateurs de comparaison

Opérateur	Dénomination	Effet	Exemple	Résultat
== **A ne pas confondre avec le signe d'affectation (=)!!**	opérateur d'égalité	Compare deux valeurs et vérifie leur égalité	x==3	Retourne *True* si X est égal à 3, sinon *False*

<	opérateur d'infériorité stricte	Vérifie qu'une variable est strictement inférieure à une valeur	x<3	Retourne *True* si X est inférieur à 3, sinon *False*
<=	opérateur d'infériorité	Vérifie qu'une variable est inférieure ou égale à une valeur	x<=3	Retourne *True* si X est inférieur ou égal à 3, sinon *False*
>	opérateur de supériorité stricte	Vérifie qu'une variable est strictement supérieure à une valeur	x>3	Retourne *True* si X est supérieur à 3, sinon *False*
>=	opérateur de supériorité	Vérifie qu'une variable est supérieure ou égale à une valeur	x>=3	Retourne *True* si X est supérieur ou égal à 3, sinon *False*
!=	opérateur de différence	Vérifie qu'une variable est différente d'une valeur	x!=3	Retourne *True* si X est différent de 3, sinon *False*

Les opérateurs logiques (booléens)

Ce type d'opérateur permet de vérifier si plusieurs conditions sont vraies :

Opérateur	Dénomination	Effet	Syntaxe
||	OU logique	Vérifie qu'une des conditions est réalisée	((condition1)||(condition2))
&&	ET logique	Vérifie que toutes les conditions sont réalisées	((condition1)&&(condition2))
!	NON logique	Inverse l'état d'une variable booléenne (retourne la valeur *True* si la variable vaut *False*, *False* si elle vaut *True*)	(!condition)

(Les opérateurs bit-à-bit)

Si vous ne comprenez pas ces opérateurs cela n'est pas important, vous n'en aurez probablement pas l'utilité. Pour ceux qui voudraient comprendre, rendez- vous aux chapitres suivants :

- compréhension du binaire
- représentation des données
- Instructions arithmétiques et logiques en assembleur

Ce type d'opérateur traite ses opérandes comme des données binaires, plutôt que des données décimales, hexadécimales ou octales. Ces opérateurs traitent ces données selon leur représentation binaire mais retournent des valeurs numériques standards dans leur format d'origine.

Les opérateurs suivants effectuent des opérations bit-à-bit, c'est-à-dire avec des bits de même poids.

Opérateur	Dénomination	Effet	Syntaxe	Résultat
&	ET bit-à-bit	Retourne 1 si les deux bits de même poids sont à 1	9 & 12 (1001 & 1100)	8 (1000)
|	OU inclusif	Retourne 1 si l'un ou l'autre des deux bits de même poids est à 1 (ou les deux)	9 | 12 (1001 | 1100)	13 (1101)
^	OU exclusif	Retourne 1 si l'un des deux bits de même poids est à 1 (mais pas les deux)	9 ^ 12 (1001 ^ 1100)	5 (0101)

(Les opérateurs de rotation de bit)

Si vous ne comprenez pas ces opérateurs cela n'est pas important, vous n'en aurez probablement pas l'utilité. Pour ceux qui voudraient comprendre, rendez- vous aux chapitres suivants :

- compréhension du binaire
- représentation des données
- Instructions arithmétiques et logiques en assembleur

Ce type d'opérateur traite ses opérandes comme des données binaires d'une longueur de 32 bits, plutôt que des données décimales, hexadécimales ou octales. Ces opérateurs traitent ces données selon leur représentation binaire mais retournent des valeurs numériques standards dans leur format d'origine.

Les opérateurs suivants effectuent des rotation sur les bits, c'est-à-dire qu'il décale chacun des bits d'un nombre de bits vers la gauche ou vers la droite. La première opérande désigne la donnée sur laquelle on va faire le décalage, la seconde désigne le nombre de bits duquel elle va être décalée.

Opérateur	Dénomination	Effet	Syntaxe	Résultat
<<	Rotation à gauche	Décale les bits vers la gauche (multiplie par 2 à chaque décalage). Les bits qui sortent à gauche sont perdus, tandis que	6 << 1 (110 << 1)	12 (1100)

		des zéros sont insérés à droite		
>>	Rotation à droite avec conservation du signe	Décale les bits vers la droite (divise par 2 à chaque décalage). Les bits qui sortent à droite sont perdus, tandis que le bit non-nul de poids plus fort est recopié à gauche	6 >> 1 (0110 >> 1)	3 (0011)
>>>	Rotation à droite avec remplissage de zéros	Décale les bits vers la droite (divise par 2 à chaque décalage). Les zéros qui sortent à droite sont perdus, tandis que des zéros sont insérés à gauche	3 >>> 1 (0011 >>> 1)	1 (0001)

Les priorités

Lorsque l'on associe plusieurs opérateurs, il faut que le compilateur sache dans quel ordre les traiter, voici donc dans **l'ordre décroissant** les priorités de tous les opérateurs :

Priorité des opérateurs

++++++++++++++++	()	[]	.					
+++++++++++++++	--	++	!	~	- (un opérande)	+ (un opérande)	(casting)	new
++++++++++++++	*	/	%					
+++++++++++++	+	-						
++++++++++++	<<	>>	>>>					
+++++++++++	<	<=	>=	>	instanceof			
++++++++++	==	!=						
+++++++++	&							
++++++++	^							
+++++++	|							
++++++	&&							
+++++	||							
++++	?	:						
+++	=	+=	-=	*=	/=	%=	&=	
++	<<=	>>=	>>>=	^=	|=			
+	,							

Qu'est-ce qu'une structure conditionnelle?

On appelle ***structure conditonnelle*** les instructions qui permettent de tester si une condition est vraie ou non. Ces structures conditionnelles peuvent être associées à des structures qui se répètent suivant la réalisation de la condition, on appelle ces structures des ***structures de boucle***

La notion de bloc

Une expression suivie d'un point-virgule est appelée instruction. Par exemple *a++;* est une instruction.
Lorsque l'on veut regrouper plusieurs instructions, on peut créer ce que l'on appelle un **bloc**, c'est-à-dire un ensemble d'instructions (suivies respectivement par des point-virgules) et comprises entre les accolades { et }.

Les instructions *if*, *while* et *for* peuvent par exemple être suivies d'un bloc d'instructions à exécuter...

L'instruction if

L'instruction if est la structure de test la plus basique, on la retrouve dans tous les langages (avec une syntaxe différente...). Elle permet d'exécuter une série d'instructions si jamais une condition est réalisée.

La syntaxe de cette expression est la suivante :

```
if (condition réalisée) {
liste d'instructions
}

else {
liste d'instructions
}
```

Remarques:

- la condition doit être entre des parenthèses
- il est possible de définir plusieurs conditions à remplir avec les opérateurs ET et OU (&& et ||)
  par exemple l'instruction suivante teste si les deux conditions sont vraies :

  ```
  if ((condition1)&&(condition2))
  ```

  L'instruction suivante exécutera les instructions si l'une ou l'autre des deux conditions est vraie

  ```
  if ((condition1)||(condition2))
  ```

- s'il n'y a qu'une instruction, les accolades ne sont pas nécessaires...
- les instructions situées dans le bloc qui suit *else* sont les instructions qui seront exécutées si la ou les conditions ne sont pas remplies

L'instruction if ... else

L'instruction *if* dans sa forme basique ne permet de tester qu'une condition, or la plupart du temps on aimerait pouvoir choisir les instructions à exécuter **en cas de non réalisation de la condition**...
L'expression *if ... else* permet d'exécuter une autre série d'instruction en cas de non-réalisation de la condition.

La syntaxe de cette expression est la suivante :

```
if (condition réalisée) {
   liste d'instructions
}

else {
         autre série d'instructions
}
```

une façon plus courte de faire un test (opérateur ternaire)

Il est possible de faire un test avec une structure beaucoup moins lourde grâce à la structure suivante, appelée opérateur ternaire :

```
(condition) ? instruction si vrai : instruction si faux
```

Remarques:

- la condition doit être entre des parenthèses
- Lorsque la condition est vraie, l'instruction de gauche est exécutée
- Lorsque la condition est fausse, l'instruction de droite est exécutée

L'instruction switch

L'instruction *switch* permet de faire plusieurs tests de valeurs sur le contenu d'une même variable. Ce branchement conditionnel simplifie beaucoup le test de plusieurs valeurs d'une variable, car cette opération aurait été compliquée (mais possible) avec des *if* imbriqués. Sa syntaxe est la suivante :

```
switch (Variable) {

case Valeur1 :
        Liste d'instructions
        break;
```

```
case Valeur2 :
        Liste d'instructions
        break;

case Valeurs... :
        Liste d'instructions
        break;

default:

        Liste d'instructions
        break;

}
```

Les parenthèses qui suivent le mot clé *switch* indiquent une expression dont la valeur est testée successivement par chacun des *case*. Lorsque l'expression testée est égale à une des valeurs suivant un *case*, la liste d'instruction qui suit celui-ci est exécuté. Le mot clé *break* indique la sortie de la structure conditionnelle. Le mot clé *default* précède la liste d'instructions qui sera exécutée si l'expression n'est jamais égale à une des valeurs.

N'oubliez pas d'insérer des instructions *break* entre chaque test, ce genre d'oubli est difficile à détecter car aucune erreur n'est signalée...

L'instruction *switch* ne fonctionne que sur des types simples (*int, short, char ou byte*) !

Les boucles

Les boucles sont des structures qui permettent d'exécuter plusieurs fois la même série d'instructions jusqu'à ce qu'une condition ne soit plus réalisée...
On appelle parfois ces structures *instructions répétitives* ou bien *itérations*.
La façon la plus commune de faire une boucle, est de créer un compteur (une variable qui s'incrémente, c'est-à-dire qui augmente de 1 à chaque tour de boucle) et de faire arrêter la boucle lorsque le compteur dépasse une certaine valeur.

La boucle for

L'instruction *for* permet d'exécuter plusieurs fois la même série d'instructions: c'est une boucle!

Dans sa syntaxe, il suffit de préciser le nom de la variable qui sert de compteur (et éventuellement sa valeur de départ, la condition sur la variable pour laquelle la boucle s'arrête (basiquement une condition qui teste si la valeur du compteur dépasse une limite) et enfin une instruction qui incrémente (ou décrémente) le compteur.

La syntaxe de cette expression est la suivante :

```
for (compteur; condition; modification du compteur) {
        liste d'instructions
}
```

Par exemple :

```
for (int i=1; i<6; i++) {
        System.out.println((char)i);

}
```

Cette boucle affiche 5 fois la valeur de *i*, c'est-à-dire 1,2,3,4,5
Elle commence à i=1, vérifie que i est bien inférieur à 6, etc... jusqu'à atteindre la valeur i=6, pour laquelle la condition ne sera plus réalisée, la boucle s'interrompra et le programme continuera son cours.

D'autre part, Java autorise la déclaration de la variable de boucle dans l'instruction *for* elle-même!

Par exemple :

```
for (int i=0; i<10; i++) {
        System.out.println((char)i);

}
```

- il faudra toujours vérifier que la boucle a bien une condition de sortie (i.e le compteur s'incrémente correctement)
- une instruction *System.out.println();* dans votre boucle est un bon moyen pour vérifier la valeur du compteur pas à pas en l'affichant!
- il faut bien compter le nombre de fois que l'on veut faire exécuter la boucle:
  - *for(i=0;i<10;i++)* exécute 10 fois la boucle (i de 0 à 9)
  - *for(i=0;i<=10;i++)* exécute 11 fois la boucle (i de 0 à 10)
  - *for(i=1;i<10;i++)* exécute 9 fois la boucle (i de 1 à 9)
  - *for(i=1;i<=10;i++)* exécute 10 fois la boucle (i de 1 à 10)

L'instruction while

L'instruction *while* représente un autre moyen d'exécuter plusieurs fois la même série d'instructions.

La syntaxe de cette expression est la suivante :

```
while (condition réalisée) {
liste d'instructions
}
```

Cette instruction exécute la liste d'instructions **tant que** (*while* est un mot anglais qui signifie *tant que*) la condition est réalisée. La condition de sortie pouvant être n'importe quelle structure conditionnelle, les risques de boucle infinie (boucle dont la condition est toujours vraie) sont grands, c'est-à-dire qu'elle risque de provoquer un plantage du navigateur!

Saut inconditionnel

Il peut être nécessaire de faire sauter à la boucle une ou plusieurs valeurs sans pour autant mettre fin à celle-ci.

La syntaxe de cette expression est "*continue;*" (cette instruction se place dans une boucle!), on l'associe généralement à une structure conditionnelle, sinon les lignes situées entre cette instruction et la fin de la boucle seraient obsolètes.

Exemple: Imaginons que l'on veuille imprimer pour x allant de 1 à 10 la valeur de 1/(x-7) ... il est évident que pour x=7 il y aura une erreur. Heureusement, grâce à l'instruction *continue* il est possible de traiter cette valeur à part puis de continuer la boucle!

```
x=1;

while (x<=10) {
          if (x == 7) {
                System.out.println("Division par zéro!");

                continue;

                }
                a = 1/(x-7);

                System.out.println((char)a);

                x++;

}
```

Il y avait une erreur dans ce programme... peut-être ne l'avez-vous pas vue :
Lorsque x est égal à 7, le compteur ne s'incrémente plus, il reste constamment à la valeur 7, il aurait fallu écrire :

```
x=1;

while (x<=10) {
```

```
            if (x == 7) {
                   System.out.println("division par 0");

                   x++;

                   continue;

            }
            a = 1/(x-7);

            System.out.println((char)a);

            x++;

}
```

Arrêt inconditionnel

A l'inverse, il peut être voulu d'arrêter prématurément la boucle, pour une autre conditon que celle précisée dans l'en-tête de la boucle. L'instruction *break* permet d'arrêter une boucle (*for* ou bien *while*). Il s'agit, tout comme *continue*, de l'associer à une structure conditionnelle, sans laquelle la boucle ne ferait jamais plus d'un tour!

Dans l'exemple de tout à l'heure, par exemple si l'on ne savait pas à quel moment le dénominateur (x-7) s'annule (bon...OK...pour des équations plus compliquées par exemple) il serait possible de faire arrêter la boucle en cas d'annulation du dénominateur, pour éviter une division par zéro!

```
for (x=1; x<=10; x++) {
a = x-7;

if (a == 0) {
System.out.println("division par 0");

break;

}

System.out.println((char) 1/a);
```

La notion d'objet

Le langage Java intègre fortement le concept objet, il s'agit donc d'un *langage orienté objet* (*LOO*). Le terme *langage orienté objet* est plus qu'une simple appellation de plus pour désigner un type de langage, il s'agit réellement d'une façon de programmer à part entière, un état d'esprit...

Pour comprendre clairement le concept d'objet, reportez-vous à la partie Programmation orientée objet!

Les classes avec Java

Pour pouvoir manipuler des objets, il est essentiel de définir des classes, c'est-à-dire définir la structure d'un objet. Cette définition avec Java se fait de la manière suivante :

```
class Nom_de_la_classe {
        // Instructions permettant de définir la classe;

}
```

Nom_de_la_classe représente bien évidemment le type d'objet désigné par la classe ou du moins le nom que vous leur attribuez.

Avec Java il n'est pas nécessaire d'ajouter un point-virgule à la fin du bloc de définition de la classe, contrairement au langage C++...

Déclaration des données membres

Jusqu'ici notre classe est vide (elle est toutefois syntaxiquement correcte), c'est-à-dire qu'elle ne contient ni données (appelées *données membres*) ni traitements (fonctions appelées *méthodes*).

Les données membres sont des variables stockées au sein d'une classe. Elles doivent être précédées de leur type et (éventuellement) d'une étiquette précisant leur portée, c'est-à-dire les classes ayant le *droit* d'y accéder.
Ces étiquettes sont au nombre de trois :

- public
- private
- protected

Pour comprendre en détail ces étiquettes, reportez-vous au chapitre sur l'encapsulation.

Ainsi, une classe comportant trois données membres peut ressembler par exemple à ceci :

```
class Voiture {
        public char Marque[32];

        private float Vitesse;

        protected int Prix;

}
```

Déclaration des méthodes

Les données membres permettent de conserver des informations relatives à la classe, tandis que les méthodes représentent les traitements qu'il est possible de réaliser avec les objets instanciés de la classe. On parle généralement de *méthodes* (parfois *fonctions membres*) pour désigner ces traitements.

La définition d'une méthode se fait en définissant le prototype et le corps de la fonction à l'intérieur de la classe en une opération

Voici donc la syntaxe de définition des méthodes :

```
class Nom_de_la_classe {
        // Instructions permettant de définir les données membres de la
classe;

        ...;

        // Instructions permettant de définir les méthodes de la classe;

        type_de_valeur_renvoyée Nom_de_la_methode(type_du_parametre1,

        // Instructions du corps de la méthode
        }
}
```

Etant donné que l'encapsulation doit permettre la protection des données membres (qui sont alors précédées de l'étiquette *private* ou *protected*), les fonctions membres doivent pouvoir servir d'interface pour manipuler les données membres. On place donc l'étiquette publique devant les fonctions membres dédiées à la manipulation des données membres. Ce système permet de garantir l'intégrité des données membres. En effet, si l'utilisateur de la classe ne peut pas modifier les données membres directement, il est obligé d'utiliser l'interface (les fonctions membres) pour les modifier, ce qui peut permettre au créateur de la classe d'effectuer des contrôles...

La notion de fonction et de méthode

On appelle *fonction* un sous-programme qui permet d'effectuer un ensemble d'instruction par simple appel de la fonction dans le **corps** du programme principal. Les fonctions permettent d'exécuter dans plusieurs parties du programme une série d'instructions, cela permet une simplicité du code et donc une taille de programme minimale. D'autre part, une fonction peut faire appel à elle-même, on parle alors de fonction récursive (il ne faut pas oublier de mettre une condition de sortie au risque sinon de ne pas pouvoir arrêter le programme...).

Une méthode est une fonction faisant partie d'une classe. Elle permet d'effectuer des traitements sur (ou avec) les données membres.

La déclaration d'une méthode

Avant d'être utilisée, une méthode doit être définie car pour l'appeler dans une classe il faut que le compilateur la connaisse, c'est-à-dire qu'il connaisse son nom, ses arguments et les instructions

qu'elle contient. La définition d'une méthode s'appelle "*déclaration*". La déclaration d'une fonction se fait selon la syntaxe suivante :

```
type_de_donnee Nom_De_La_Methode(type1 argument1, type2 argument2, ...) {
                          liste d'instructions
}
```

Remarques:

- type_de_donnee représente le type de valeur que la méthode est sensée retourner (char, int, float,...)
- Si la méthode ne renvoie aucune valeur, on la fait alors précéder du mot-clé *void*
- Une méthode doit obligatoirement porter un type de retour (sauf dans le cas des constructeurs)
- le nom de la méthode suit les mêmes règles que les noms de variables :
  - le nom doit commencer par une lettre
  - un nom de méthode peut comporter des lettres, des chiffres et les caractères _ et $ (les espaces ne sont pas autorisés!)
  - le nom de la méthode, comme celui des variables est sensible à la casse (différenciation entre les minuscules et majuscules)
- Les arguments sont facultatifs, mais s'il n'y a pas d'arguments, les parenthèses doivent rester présentes
- Il ne faut pas oublier de refermer les accolades

- `Le nombre d'accolades ouvertes (méthode, boucles et autres structures) doit être égal au nombre d'accolades fermées!`
- `La même chose s'applique pour les parenthèses, les crochets ou les guillemets!`

Une fois cette étape franchie, votre méthode ne s'exécutera pas tant que l'on ne fait pas appel à elle quelque part dans la classe!

Appel de méthode

Pour exécuter une méthode, il suffit de faire appel à elle en écrivant son nom (une fois de plus en respectant la casse) suivie d'une parenthèse ouverte (éventuellement des arguments) puis d'une parenthèse fermée :

```
Nom_De_La_méthode();
```

Remarques:

- le point virgule signifie la fin d'une instruction et permet au navigateur de distinguer les différents blocs d'instructions
- si jamais vous avez défini des arguments dans la déclaration de la méthode, il faudra veiller à les inclure lors de l'appel de la méthode (le même nombre d'arguments séparés par des virgules!)

```
Nom_De_La_méthode(argument1, argument2);
```

Java impose qu'une fonction qui ne retourne aucune valeur soit précédée du type *void*!

Les arguments d'une méthode

Il est possible de passer des arguments (appelés aussi *paramètres*) à une méthode, c'est-à-dire lui fournir une valeur ou le nom d'une variable afin que la méthode puisse effectuer des opérations sur ces arguments ou bien grâce à ces arguments.
Le passage d'arguments à une méthode se fait au moyen d'une liste d'arguments (séparés par des virgules) entre parenthèses suivant immédiatement le nom de la méthode.

Le nombre et le type d'arguments dans la déclaration, le prototype et dans l'appel doit correspondre, au risque, sinon, de génerer une erreur lors de la compilation...

Un argument peut être :

- une constante
- une variable
- une expression
- une autre méthode retournant une valeur

Renvoi d'une valeur par une méthode

La méthode peut renvoyer une valeur (et donc se terminer) grâce au mot-clé *return*. Lorsque l'instruction *return* est rencontrée, la méthode évalue la valeur qui la suit, puis la renvoie au programme appelant (classe à partir de laquelle la méthode a été appelée).

Une méthode peut contenir plusieurs instructions *return*, ce sera toutefois la première instruction *return* rencontrée qui provoquera la fin de l'exécution de la méthode et le renvoi de la valeur qui la suit.

La syntaxe de l'instruction return est simple :

```
return valeur_ou_variable;
```

Le type de valeur retourné doit correspondre à celui qui a été précisé dans la définition

La surcharge de méthode

Un des apports les plus intéressants du Java, hormis l'ajout du concept objet, est la possibilité d'appeler plusieurs méthodes avec le même nom, à condition que leurs arguments diffèrent (en type et/ou en nombre).

Ce principe est appelé surcharge de méthode. Il permet de donner le même nom à des méthodes comportant des paramètres différents et simplifie donc l'écriture de méthodes sémantiquement similaires sur des paramètres de type différent.

En effet, une méthode est déterminé par ce que l'on appelle sa *signature*, c'est-à-dire :

- son nom
- ses paramètres

Il est ainsi possible de définir une méthode réalisant la même opération sur des variables différentes en nombre ou en type :

```
int somme( int p1, int p2){
return (p1 + p2);

}

float somme( float p1, float p2){
return (p1 + p2);

}

float somme( float p1, float p2, float p3){
return (p1 + p2 + p3);

}

int somme( float p1, int p2){
return (int(p1) + p2);
```

La création d'objets

Un objet est une instance de classe, la création d'objets s'appelle donc l'*instanciation*. Cette instanciation se fait grâce à l'opérateur *new* suivi du nom de la classe à instancier et de parenthèses contenant les paramètres d'instanciation (parenthèses vides s'il n'y a pas de paramètres). Soit la classe *Personne* suivante :

```
class Personne{
        int age;

}
```

L'instanciation de cette classe (rudimentaire et actuellement inutile...) se fait de la façon suivante :

```
new Personne();
```

Les références (handles)

Dans l'état actuel des choses (la simple création d'un objet avec le mot clé *new*), l'objet nouvellement créé est tout simplement inutilisable, car une fois que l'on a créé l'objet il n'y a aucune façon de le désigner (*Personne* représente le nom de la classe et non celui d'une de ces instanciations).

Il est heureusement possible d'identifier chaque objet par un nom grâce à un élément appelé *référence* (en anglais *handle*, littéralement *poignée*). On ne peut toutefois pas dire qu'un *handle* soit le nom d'un objet, étant donné qu'un objet peut posséder plusieurs *handles* (un *handle* ne peut par contre désigner qu'un seul objet). Le concept des *handles* est très proche de celui des pointeurs en langage C/C++, il est toutefois beaucoup plus simple à comprendre et à mettre en oeuvre.

Pour créer une *référence* d'objet il suffit de faire précéder le nom que l'on donne à un objet par le nom de la classe que l'on instancie, par exemple une référence de l'objet Toto, instance de la classe Personne, peut être défini de la manière suivante :

```
Personne Toto;
```

La *référence* Toto ne pointe pour l'instant vers aucun objet. Pour ce faire, il suffit de l'initialiser, c'est-à-dire en réalité de lui affecter le résultat de la création de l'objet renvoyé par l'opérateur *new*. Avec l'exemple précédent cela pourrait s'écrire de la façon suivante :

```
Personne Toto;

Toto = new Personne();
```

L'affectation de l'opérateur *new* à la référence ne signifie en aucun cas que l'on stocke l'objet dans la référence, cela indique que la référence pointe vers l'objet nouvellement créé!

Les deux lignes précédentes peuvent être regroupées en un seul de la façon suivante :

```
Personne Toto = new Personne();
```

Accéder aux données membres d'un objet

L'accès aux données membres d'un objet se fait grâce au nom de la référence vers l'objet, suivi d'un point puis du nom de la donnée membre. Par exemple :

```
Nom_du_handle.Nom_de_la_donnee_membre = Valeur;
```

Si jamais la donnée membre est une référence vers un autre objet, on peut accéder à ses données membres par l'intermédiaire de l'objet en cours :

```
Nom_du_handle.Nom_donnee_membre_objet.Nom_donnee_membre_de_l_autre_objet =
Valeur;
```

Accéder aux méthodes d'un objet

L'accès aux méthodes d'un objet se fait comme pour l'accès aux données membres, c'est-à-dire par un point. La méthode est suivie de parenthèses, contenant les paramètres, si il y'en a. L'accès à une fonction membre se fait donc de la façon suivante :

```
Nom_du_handle.Nom_donnee_fonction_membre(parametre1,parametre2,...);
```

Le mot clé *this*

Le mot clé *this* permet de désigner l'objet dans lequel on se trouve, c'est-à-dire que lorsque l'on désire faire référence dans une fonction membre à l'objet dans lequel elle se trouve, on utilise *this*.

L'objet courant *this* est en réalité une variable système qui permet de désigner l'objet courant. Cette variable est passée en tant que paramètre caché de chaque fonction membre.

Ainsi, lorsque l'on désire accéder à une donnée membre d'un objet à partir d'une fonction membre du même objet, il suffit de faire précéder le nom de la donnée membre par *this.*. Par exemple :

```
class Toto{
        int age;

        char sexe[16];

        void DefineTotoAge(int age){
                this.age = age;

        }
}
```

En réalité, lorsque l'on donne des noms différents aux données membres et aux variables utilisées dans les fonctions membres, la variable *this* est implicite, cela signifie que l'on est pas obligé de mettre ***this.*** devant chaque donnée membre.

Dans le cas ou l'on désire une fonction membre qui retourne un pointeur vers l'objet dans lequel elle se trouve, la variable *this* est indispensable :

```
class Toto{
        int age;

        char sexe[16];

        void DefineTotoAge(int age){
                this.age = age;

        }
```

```
        Toto * RetourneToto(){
        return this;

}

}
```

Le constructeur: une méthode particulière

Pour instancier une classe, c'est-à-dire créer un objet à partir d'une classe, il s'agit d'utiliser l'opérateur *new*.
En réalité l'opérateur *new*, lorsqu'il est utilisé, fait appel à une méthode spéciale de la classe: le **constructeur**.

Le rôle du constructeur est de déclarer et de permettre d'initialiser les données membres de la classe, ainsi que de permettre différentes actions (définies par le concepteur de la classe) lors de l'instanciation.

Un constructeur se définit comme une méthode standard, mais ne renvoie aucune valeur.
Ainsi, le constructeur d'un objet porte le même nom que la classe et ne possède aucune valeur de retour (même pas *void*).

- un constructeur porte le même nom que la classe dans laquelle il est défini
- un constructeur n'a pas de type de retour (même pas *void*)
- un constructeur peut avoir des arguments
- la définition d'un constructeur n'est pas obligatoire lorsqu'il n'est pas nécessaire

La définition de cette fonction membre spéciale n'est pas obligatoire (si vous ne souhaitez pas initialiser les données membres par exemple) dans la mesure où un *constructeur par défaut* (appelé parfois *constructeur sans argument*) est défini par le compilateur Java si la classe n'en possède pas.

Voyons sur un exemple comment se déclare un constructeur :

```
class Toto{
int age;

char sexe;

float taille;

Toto(int age, char sexe, float taille){
```

```
        this.age = age;

        this.sexe = sexe;

        this.taille = taille;

  }
}
```

Définir plusieurs constructeurs

Comme pour n'importe quelle méthode, il est possible de surcharger les constructeurs, c'est-à-dire définir plusieurs constructeurs avec un nombre/type d'arguments différents. Ainsi, il sera possible d'initialiser différemment un même objet, selon la méthode de construction utilisée.

Imaginons par exemple que pour l'exemple précédent on veuille pouvoir définir le sexe de Toto grâce à un entier valant 0 ou 1, ainsi qu'avoir la possibilité de passer en paramètre la lettre 'M' ou 'F', on peut alors définir deux constructeurs pour lesquels le type du second argument sera différent. De plus, on va montrer de quelle manière il est possible de contrôler le caractère entré en paramètre :

```
class Toto{
int age;

char sexe;

float taille;

Toto(int age, char sexe, float taille){
        this.age = age;

        if ((sexe=='M')||(sexe=='F')) {
        this.sexe = sexe;

        }
        else System.out.println("Erreur d'initialisation");

        this.taille = taille;

}

Toto(int age, int sexe, float taille){
        this.age = age;

        switch (sexe) {
                case 0 :
                        this.sexe = 'F';

                        break;

                case 1:
                        this.sexe = 'M';
```

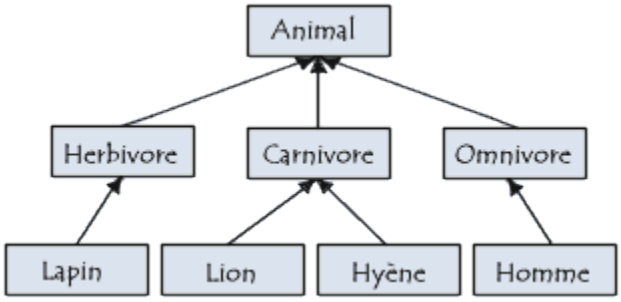
```
                        break;

                default :
                        System.out.println("Erreur d'initialisation");

                        break;

                }
        this.taille = taille;

}

}
```

La notion d'héritage

Le concept d'héritage est un des concepts les plus importants de la programmation orientée objet, car il conditionne irréversiblement la façon selon laquelle un code Java est écrit. L'héritage est un mécanisme permettant de créer une nouvelle classe à partir d'une classe existante en lui proférant ses propriétés et ses méthodes.

Ainsi, pour définir une nouvelle classe, il suffit de la faire hériter d'une classe existante et de lui ajouter de nouvelles propriétés/méthodes.

De cette façon, les classes héritées forment une hiérarchie descendante, au sommet de laquelle se situe la classe de base (superclasse). On appelle également la classe héritée la *sous-classe* et la classe parente la *super-classe*.



Avec Java, l'arborescence est stricte, c'est-à-dire qu'une classe donnée ne peut possèder qu'une seule superclasse (l'héritage est dit *simple* contrairement à des langages comme le C++, pour lesquels un héritage dit *multiple* est possible). Par contre, une classe peut possèder un nombre illimité de sous-classes.

Lors de l'instanciation d'une classe, celle-ci hérite de (cela signifie qu'elle reçoit) l'ensemble des propriétés et des méthodes de sa superclasse, qui elle-même hérite d'une éventuelle superclasse, etc.

Par conséquent le travail principal du programmeur Java consiste à concevoir une hiérarchie de classe, c'est-à-dire organiser les classes de telles façon que les propriétés et méthodes communes à plusieurs classes soient placées dans une superclasse.

Grâce à ce système organisationnel, il est aisé de réutiliser des composants existants et de leur ajouter un comportement. De plus, la modification de la superclasse implique la modification automatique de toutes les sous-classes. Pour plus d'informations sur l'héritage, consulter la section "programmation orientée objet"

L'héritage avec Java

Une classe se définit tout simplement grâce au mot-clé *class* de la manière suivante par exemple :

```
class Ccm {
        // Corps de la classe
}
```

Par défaut une classe hérite de la "super-superclasse" nommée *Object*. Pour faire hériter une classe d'une superclasse, Java fournit le mot-clé **extends**, c'est-à-dire que les quelques lignes de code ci-dessus sont équivalentes à:

```
class Ccm extends Object {
        // Corps de la classe
}
```

Le mot-clé super

Au même titre que le mot-clé *this* permet de faire référence à l'objet en cours, le mot-clé *super* permet de désigner la superclasse, c'est-à-dire qu'à l'aide du mot-clé *super*, il est possible de manipuler les données membres et les méthodes de la superclasse.

Pour manipuler une proriété de la superclasse, il suffit d'utiliser la syntaxe suivante :

```
super.nom_de_la_propriete
```

De la même façon, pour manipuler une méthode de la superclasse, il suffit d'utiliser la syntaxe suivante :

```
super.nom_de_la_methode()
```

Enfin, il est possible de faire appel au constructeur de la classe parente en fournissant des arguments au mot-clé *super* de la façon suivante :

```
super(parametre1,parametre2,...)
```

Remplacer une méthode existante de la superclasse

Lorsqu'une classe hérite de sa superclasse, elle hérite de ses méthodes, c'est-à-dire qu'elle possède les mêmes méthodes que sa superclasse.

Il est possible de redéfinir totalement une méthode en la redéfinissant (mêmes nombres et types d'arguments que la méthode de la superclasse). De cette façon, la méthode originale est ignorée au profit de sa redéfinition.

Ajouter un comportement à une méthode de la superclasse

La redéfinition d'une méthode consiste à réécrire totalement la méthode initiale. Il est parfois utile de simplement ajouter un comportement au comportement original, pour cela l'utilisation du mot-clé *super* s'avère nécessaire. En appelant le constructeur de la superclasse dans le corps de la méthode qui la redéfinit, et en ajoutant de nouvelles instructions, la méthode originale est dotée de comportements supplémentaires.

La notion d'accessibilité

Le concept d'accessibilité (généralement appelé *encapsulation*) définit la possibilité qu'a le concepteur d'une classe de restreindre l'accès à certaines données, ou plus généralement à certains éléments (méthodes, classes, ...).

En effet, les classes sont prévues pour être diffusées (parfois de façon payante) afin de pouvoir être réutilisées par certains programmeurs. Or il est généralement souhaitable que les utilisateurs de la classe se servent de celle-ci de la manière de laquelle le concepteur a prévu qu'elle le soit. Ainsi, il est possible de restreindre l'accès à (ou bien l'instanciation de) certains éléments en leur associant une étiquette.

La protection des données membres

l'encapsulation consiste donc à définir des étiquettes pour les données membres et les méthodes afin de préciser si celles-ci sont accessibles à partir d'autres classes ou non...

De cette manière, des données membres portant l'étiquette *private* ne peuvent pas être manipulées directement par les méthodes des autres classes. Ainsi, pour pouvoir manipuler ces données membres, le créateur de la classe (vous en l'occurrence) doit prévoir des méthodes spéciales portant l'étiquette *public*, permettant de manipuler ces données.

- Les fonctions membres permettant d'accéder aux données membres sont appelées **accesseurs**, parfois *getter* (appelation d'origine anglophone)
- Les fonctions membres permettant de modifier les données membres sont appelées **mutateurs**, parfois *setter* (appelation d'origine anglophone)

On appelle généralement "**interface**" l'ensemble des méthodes permettant à l'utilisateur de manipuler une classe.

Les niveaux d'accès

Java définit quatre niveaux d'accès pour les variables d'instances (données membres) et les méthodes :

- **public** : un élément *public* est accessible de partout et sans aucune restriction. Certaines classes (comme la classe principale *main*) doivent obligatoirement être déclarées publiques (pour pouvoir exécuter l'application...)
- **protected** : un élément *protected* (*protégé*) est accessible uniquement aux classes d'un package et à ses classes filles
- **private** : un élément *private* (*privé*) est accessible uniquement au sein de la classe dans laquelle il est déclaré. Ces éléments ne peuvent être manipulés qu'à l'aide de méthode spécifiques appelés accesseur et mutateur
- **"friendly"** : un élément est *friendly* **par défaut** (cette appellation n'est pas officielle et est empruntée au langage C++) est accessible uniquement aux classes d'un package et à ses classes filles.

La notion d'accesseur

Un accesseur est une méthode permettant de récupérer le contenu d'une donnée membre protégée. Un accesseur, pour accomplir sa fonction :

- doit avoir comme type de retour le type de la variable à renvoyer
- ne doit pas nécessairement posséder d'arguments

Une convention de nommage veut que l'on fasse commencer de façon préferrentielle le nom de l'accesseur par le préfixe *get*, afin de faire ressortir sa fonction première.

La syntaxe d'un accesseur réduit à sa plus simple expression ressemble donc à ceci :

```
public class MaClasse{
        public static void main(String[] argv){
                new MaSecondeClasse();

        }
}

class MaSecondeClasse{
                private TypeDeMaVariable maVariable;

                public TypeDeMaVariable getMaVariable(){
                        return maVariable;

                }
}
```

Sur l'exemple précédent, l'accesseur minimal de la donnée membre *age* pourrait être le suivant :

```
class Toto{
        private int age;
```

```
        public int getAge(){
                return age;

        }
}
```

La notion de mutateur

Un mutateur est une méthode permettant de modifier le contenu d'une donnée membre protégée. Un mutateur, pour accomplir sa fonction :

- doit avoir comme paramètre la valeur à assigner à la donnée membre. Le paramètre doit donc être du type de la donnée membre
- ne doit pas nécessairement renvoyer de valeur (il possède dans sa plus simple expression le type *void*)

Une convention de nommage veut que l'on fasse commencer de façon préferrentielle le nom du mutateur par le préfixe *set*.

La syntaxe d'un mutateur réduit à sa plus simple expression ressemble donc à ceci :

```
class MaClasse{
        private TypeDeMaVariable maVariable;

        public  void setMaVariable(TypeDeMaVariable maValeur){
                maVariable = maValeur;

        }
}
```

Sur l'exemple précédent, le mutateur minimal de la donnée membre *age* pourrait être le suivant :

```
class Toto{
        private int age;

        public void setAge(int nouveauAge){
                age = nouveauAge;

        }
}
```

L'intérêt principal d'un tel mécanisme est le contrôle de la validité des données membres qu'il procure. En effet, il est possible (et même conseillé) de tester la valeur que l'on assigne à une donnée membre, c'est-à-dire que l'on effectue un test de validité de la valeur de l'argument avant de l'affecter à la donnée membre. Le mutateur ci-dessus peut par exemple vérifier si l'âge de Toto est correct (on considérera pour cela que Toto ne peut pas vivre plus de 200 ans...c'est avec des hypothèses telles que celle-ci que peuvent apparaître des bogues...ah les progrès de la génétique!).

```
class Toto{
        private int age;
```

```
        public int setAge(int nouveauAge){
        if (nouveauAge < 200) {
                age = nouveauAge;

                return 1;

        }
        else return 0;

        }
}
```

Introduction aux packages

La programmation Java consiste à créer des classes. Or, étant donné qu'un programme est généralement développé par une équipe de programmeurs, c'est-à-dire plusieurs personnes, le fait de concaténer (mettre bout à bout) des classes dans un fichier est loin d'être satisfaisant. C'est pour cette raison que Java propose l'utilisation de **packages** (comparables aux bibliothèques du langage C++/C)

Ainsi, un **package** est une unité (un fichier) regroupant des classes. Pour créer un tel package, il suffit de commencer le fichier source contenant les classes à regrouper par l'instruction *package* suivi du nom que l'on désire donner au package. dès lors, toutes les classes contenues dans le fichier feront partie du package...

L'organisation hiérarchique

Les packages sont organisés sous forme de hiérarchie, c'est-à-dire qu'il est possible de les imbriquer, pour cela Java impose que l'emplacement des fichiers sources (donc des packages) correspondent à la hiérarchie des packages, c'est pour cela qu'un package portant le nom MonPackage doit être stocké dans un répertoire du même nom. De plus, pour que le compilateur puisse trouver le package, il est essentiel qu'il "connaisse" l'emplacement du package. Pour cela Java utilise une variable d'environnement (dans le même ordre d'idée que la variable d'environnement PATH) appelée **classpath** donnant la liste des chemins d'accès aux classes. Par défaut le compilateur (ainsi que la machine virtuelle) recherchent les classes dans le répertoire courant et le répertoire des classes standards.

Pour définir la variable d'environnement **classpath**

- sous **Linux**, il faut utiliser la commande suivante :

  ```
  export CLASSPATH = <repertoire>:<repertoire>;...
  ```

  par exemple :

```
export CLASSPATH =
/home/jeff/java:/usr/lib/jdkx.x.x/lib/classes.zip:...
```

- *sous **Windows**, il faut utiliser la commande suivante :*

  ```
  SET CLASSPATH = <repertoire>;<repertoire>;...
  ```

  par exemple :

  ```
  SET CLASSPATH = c:\java;c:\jdkx.x.x\lib\classes.zip;...
  ```

L'instruction import

Pour pouvoir accèder aux classes d'un package à partir d'une classe ne faisant pas partie du package, il suffit de donner le chemin d'accès relatif à la classe lorsque vous faîtes appel à elle. Ainsi, pour faire appel à la classe MaClasse du package MonPackage, il suffit d'écrire :

```
MonPackage.MaClasse
```

*Toutefois il s'avére vite embarrassant d'avoir à écrire constamment le chemin d'une classe, ainsi il est généralement pratique d'utiliser l'instruction **import**, suivie du chemin de la classe :*

```
import MonPackage.MaClasse;
```

Il est également possible d'importer toutes les classes d'un package par la syntaxe :

```
import MonPackage.*;
```

Voilà ce à quoi ressemble une application Java utilisant des packages (c'est-à-dire tous ou presque car vous aurez à utiliser des packages fournis avec le JDK) :

```
import java.awt.Button;   // On importe la classe Button
import java.rmi.*;     // On importe toutes les classes RMI
class Exemple {
  static void main(String argv[]){
        Rect MonRect =new Rectangle();

        ...
  }
}
```

Contrairement à ce que l'on pourrait croire, l'instruction import MonPackage.; permet de rendre accessible uniquement les classes contenues dans le package MonPackage et non à l'ensemble des packages dont le nom commence par MonPackage, c'est-à-dire que la classe MonPackage.pack1.MaClasse ne sera pas accessible...*

Convention de dénomination des packages

Etant donné la structure hiérarchique des packages et le nombre considérables de packages créés par des développeurs du monde entier, il est essentiel d'éviter de donner le même nom à

des packages différents.
*Ainsi Java propose une dénomination standard des packages. Cette appellation standard consiste à donner un nom au package et de "l'allonger" par le nom de la société, ou du concepteur des classes qu'il contient. Ainsi, un package soundstuffs développé par **CCM** aurait pour dénomination net.commentcamarche.soundstuffs.*

packages et accessibilité

L'accessibilité des données membres et des méthodes d'une classe existe aussi au travers des packages. Ainsi, une classe n'est par défaut visible qu'à partir des classes faisant partie du même package. Or l'intérêt d'un package est justement de pouvoir être utilisé à partir de classes déclarées dans un autre fichier, c'est la raison pour laquelle les classes d'un package destinées à être exportées doivent impérativement être déclarées publiques...

Packages accessibles par défaut

Java définit deux packages par défaut, c'est-à-dire un ensemble de classes auxquel il est possible de faire référence sans spécifier le chemin d'accès (que ce soit avec une instruction import ou en faisant précéder le nom par la hiérarchie de packages). Ces packages sont :

1. *Le **package par défaut**: il s'agit de toutes les classes ne faisant pas partie explicitement d'un package et situées dans les répertoires accessibles (par défaut ou dans la variable classpath)*
2. *Le **package java.lang** contenant les classes standards telles que System ou Math*

Les fichiers .jar

Avec la croissance des réseaux, les applications sont vouées à voyager, il faut donc s'assurer :

- *qu'elles restent intactes*
- *que tous les éléments (packages) de l'application sont présents*
- *de rendre l'application la moins gourmande en espace*

*Ainsi, Java propose l'utilitaire **jar** dans le JDK, un utilitaire permettant de rassembler les différentes classes (fichiers .class) d'une application au sein d'une archive compressée, appelé package, afin d'en assurer l'intégrité et la taille.*

Grâce à cet utilitaire, il est possible d'appeler à partir d'une page Web l'ensemble des classes d'une applet en faisant uniquement référence à l'archive (dont l'extension est .jar).

*La syntaxe de **jar** est proche de celle de **tar** sous Unix :*

- *Pour créer l'archive la commande est la suivante :*

```
jar cvf MonArchive.jar *.class
```

- *Pour décompresser l'archive la commande est :*

  ```
  jar xvf MonArchive.jar *.class
  ```

- *Lorsque vous utilisez une archive Jar sur Internet, toutes les classes de celle-ci seront téléchargées en même temps, veillez donc à y inclure uniquement les classes nécessaires pour une application donnée*
- *L'utilitaire MoaJar permet de manipuler les archives **Jar** à l'aide d'une interface graphique*

La nécessité de détecter les erreurs

Tout programme comporte des erreurs, même si celui-ci semble fonctionner à merveille. Chaque programmeur essaye de réduire au minimum le nombre d'erreurs, mais toutes les erreurs ne peuvent pas forcément être prévues.

Les erreurs syntaxiques sont la plupart interceptées lors de la compilation, mais il reste souvent des erreurs "imprévisibles".
Ces erreurs se produisent généralement de façon **exceptionnelle**, c'est-à-dire suite à une action de l'utilisateur, ou de l'environnement. La solution consiste donc dans un premier temps à "*prévoir les erreurs imprévisibles*" en mettant en place un système de codes d'erreurs, c'est-à-dire un système permettant de retourner des valeurs spécifiques pour signaler un fonctionnement anormal de l'application (souvent *-1* ou *NULL*).

Toutefois cette solution est loin d'être satisfaisante car :

- elle rend difficile l'écriture de programmes
- elle rend difficile la lecture du code source, à cause d'une embrication de test conditionnels (*if .. else*)
- le code d'erreur retournée peut-être confondue avec la valeur retournée par la fonction... (-1 peut très bien être la valeur de retour d'une fonction)

La notion d'exception

Pour traiter les erreurs, Java propose un mécanisme qualifié d'exception, consistant à effectuer les instructions dans un bloc d'essai (le bloc *try*) qui surveille les instructions. Lors de l'apparition d'une erreur, celle-ci est </ital>lancée</ital> dans un bloc de traitement d'erreur (le bloc *catch*, appelé *handler d'exception*) sous forme d'un objet appelé **Exception**.

Le bloc de traitement d'erreur va *lever* (il s'agit du terme technique, certains diront aussi *intercepter*) l'exception. Le *handler d'exception* peut alors traiter l'erreur (en signalant l'erreur par exemple, ou en attendant avant de réessayer, ...) puis lancer à nouveau l'exception vers un bloc de plus haut niveau.

Il est important de souligner que la transmission des exceptions est d'abord *implicite*. C'est bien évidemment pour cette raison que le concept d'exception est un réel progrès pour l'écriture d'un code facile à lire.

Les exceptions par la pratique

La gestion des exceptions avec Java consiste à définir au sein d'une méthode une clause "*try{}*" contenant les instructions qui risquent de provoquer une exception et de la faire suivre immédiatement par une clause "*catch{}*" contenant comme paramètre l'exception attendue précédée de son type (pour une erreur mathématique ce sera *ArithmeticException*) et dont le contenu sera une liste d'instruction à exécuter lorsque l'exception se produira.

Voici la syntaxe type d'une classe gérant des exceptions :

```
class Nom_de_la_classe {
 public static void main(String[] args) {
 // Instructions inoffensives (affectations, ...);
 try {
 // Instructions susceptibles de provoquer des erreurs;
 }
 catch (TypeException e) {
 // Instructions de traitement de l'erreur;
 }
 // Instructions si aucune erreur est apparue;
 }
}
```

- Aucune instruction ne doit se trouver entre le bloc *try{}* et le bloc *catch{}*
- Les blocs *try{}* et *catch{}* peuvent chacun contenir plusieurs instructions</ital>

Comment les exceptions sont-elles traitées?

Lorsque le programme rencontre une exception dans un bloc *try{}*, une exception est instanciée puis lancée. L'interpréteur cherche un bloc *catch(){}* à partir de l'endroit où l'exception a été créée en cherchant vers le bas. S'il ne trouve aucun bloc *catch{}*, l'exception est lancée dans le bloc de niveau supérieur, ainsi de suite jusqu'au bloc de la classe qui par défaut enverra l'exception au handler de l'interpréteur. Celui-ci émettra alors un message d'alerte standard pour le type d'exception.

Si jamais un bloc *catch{}* est trouvé, celui-ci gèrera l'exception à sa façon (ou plutôt à la votre) et l'exception ne sera pas envoyée au handler de l'interpréteur.

Si par contre on désire que l'exception soit traité par les blocs de niveaux supérieurs, il suffit d'inclure à la fin de la série d'instructions contenues dans le bloc *catch{}* une clause *throw*, suivie du type de l'exception entre parenthèse puis du nom de l'exception (son *handle* pour utiliser un terme exact). Ainsi l'exception continuera son chemin...

```
class Nom_de_la_classe {
 public static void main(String[] args) {
 // Instructions inoffensives (affectations, ...);
 try {
 // Instructions susceptibles de provoquer des erreurs;
 }
 catch (TypeException e) {
 // Instructions de traitement de l'erreur;
 throw (TypeException)e;
 }
 // Instructions si aucune erreur est apparue;
```

```
  }
}
```

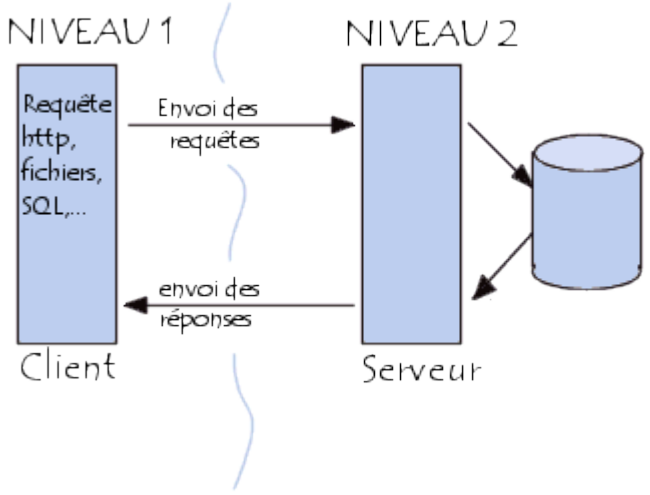
Introduction à JDBC

La technologie JDBC (*Java DataBase Connectivity*) est un ensemble de classes permettant de développer des applications capables de se connecter à des serveurs de bases de données (SGBD).

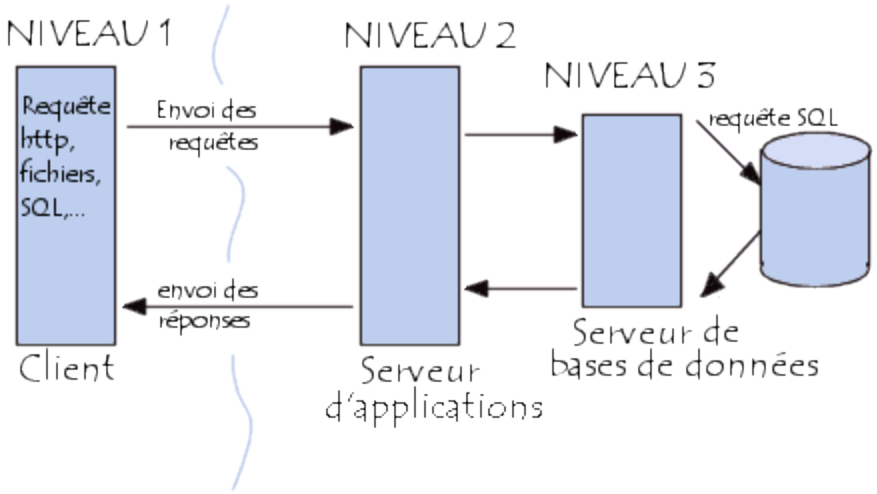
L'accès aux bases de données avec JDBC

Dans un système client/serveur, l'accès aux bases de données avec JDBC peut s'effectuer selon un modèle à deux couches ou bien un modèle à trois couches.

Pour le modèle à deux couches, une application Java est intimement liée avec une base de données. A cet effet, il faut bien évidemment disposer, pour la base de données concernée, d'un pilote JDBC adéquat. Les instructions SQL sont directement envoyées à la base, cette dernière renvoyant les résultats par un biais tout aussi direct. La base de données peut être exécutée sur la machine locale (celle sur laquelle l'application Java fonctionne) ou bien sur tout autre ordinateur du réseau (Intranet ou Internet).



Dans le modèle à 3 couches, une troisième couche (le serveur d'application) vient s'intercaler entre l'application Java et la base de données. Les instructions SQL et les résultats livrés en retour y transitent. Ce modèle présente l'avantage d'intégrer dans cette couche un contrôle d'accès.



Les types de pilotes JDBC

Les pilotes actuels sont classés en quatre catégories :

- les passerelles JDBC-ODBC, avec le pilote ODBC appartenant à une base de données. Le pilote convertit les appels de données Java en appel ODBC valide, et les exécute ensuite à l'aide du pilote ODBC
- un mélange de pilotes natifs et de pilotes Java. Les appels JDBC sont convertis en appels natifs pour le serveur de bases de données (Oracle, Sybase, ou autres).
- les pilotes convertissant les appels JDBC en un protocole indépendant du SGBD. Un serveur convertit ensuite ceux-ci dans le protocole SGBD requis (modèle à 3 couches)
- les pilotes JDBC, convertissant les appels JDBC directement en un protocole réseau exploité par le SGBD. Cette solution est à préconiser dans le cadre d'un intranet

A l'heure actuelle, les bases de données disposent pratiquement toutes d'un pilote ODBC, étant donné que nombre d'entre-elles ont été développées à partir d'interfaces Microsoft.

La technologie ODBC

ODBC signifie *Open Database Connectivity*. Il s'agit d'un format propriétaire défini par Microsoft permettant la communication entre des clients pour bases de données fonctionnant sous Windows et les SGBD du marché.

Etant donné qu'il existe différents types de bases de données, et que chacune d'entre-elles possède sa propre façon de traiter les requêtes SQL et de renvoyer les résultats, ODBC permet d'obtenir des résultats identiques quel que soit le type de base de données, sans avoir à modifier le programme d'appel qui transmet la requête.

ODBC n'est toutefois pas la solution miracle, étant donné d'une part qu'il ne s'agit pas d'une technologie libre (Microsoft la fait évoluer à son gré), d'autre part qu'il ne s'agit que d'une interface "alternative" à celle fournie par les constructeurs et compliquée à maîtriser.

La passerelle JDBC-ODBC

Afin d'illustrer la connexion à une base de données, nous allons nous connecter à une base Access par l'intermédiaire de la passerelle ODBC-JDBC, en abordant toutes les étapes de la connexion à une base de données.

Après création de la base sous Access, il faut dans un premier temps intégrer la base dans l'administrateur de source de données ODBC (*Panneau de configuration / source de données*). Les étapes de création de la source ODBC sont :

- la sélection du pilote pour la base de données concernée (dans notre cas le pilote *Microsoft Access Driver*)
- la définition du nom de la base et de son chemin d'accès
- la définition du nom de la source de données et du mot de passe

Connexion à la base de données

L'API (*Application Programming Interface*) JDBC, c'est-à-dire la bibliothèque de classes JDBC, se charge de trois étapes indispensables à la connexion à une base de données :

- la céation d'une connexion à la base
- l'envoi d'instructions SQL
- l'exploitation des résultats provenant de la base

Connexion à la base de données

Tous les objets et les méthodes relatifs aux bases de données sont présentes dans le package *java.sql*, il est donc indispensable d'importer *java.sql.** dans tout programme se servant de la technologie JDBC.

Pour se connecter à une base de données déclarée dans l'administrateur ODBC, il est essentiel de charger dans un premier temps le pilote JDBC-ODBC (appelé *pont JDBC-ODBC*) :

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Cette instruction charge le pilote et crée une instance de cette classe. Pour se connecter à une base de données particulière, il s'agit ensuite de créer une instance de la classe *Connection* en indiquant la base de données à charger à l'aide de son URL

```
String url = "jdbc:odbc:base_de_donnees";

Connection con = DriverManager.getConnection(url);
```

Le nom de la base de données (ici *base_de_donnees*) étant celle déclarée dans le panneau de configuration ODBC. La syntaxe de l'URL peut varier légèrement selon le type de la base de données. Il s'agit généralement d'une adresse de la forme :
`jdbc:sousprotocole:nom`

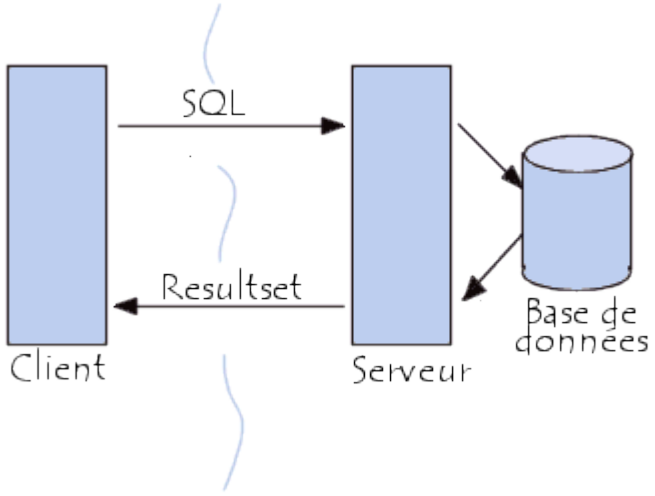
Exécution d'une requête SQL

Pour exécuter une requête SQL, il s'agit dans un premier temps de créer un objet *Statement*, pouvant être obtenu à partir de l'objet *Connection*. Un objet *ResultSet* permettra de récupérer les données en provenance de l'objet *Statement*.

```
String query = "SELECT * FROM Ma_Table;";

ResultSet results;

try {
        Statement stmt = con.createStatement();

        results = stmt.executeQuery(query);

}

catch(Exception e){
        System.out.println("exception du a la requete");

}
```

Accès aux données

Une fois la connexion à la base de données établie, il est possible de demander des informations sur le nom des tables et le contenu de chaque colonne, ainsi que d'exécuter des requêtes SQL afin de récupérer des informations, d'en ajouter ou bien de les modifier.



Les objets utilisables pour obtenir des informations sur la base sont :

- **DataBaseMetaData**: Un objet contenant les informations sur la base de données en général (des *métadonnées*), c'est-à-dire le nom de la table, les index, la version de la base, ...
- **ResultSet**: Un objet contenant les informations sur une table ou le résultat d'une requête. L'accès aux données se fait colonne par colonne, mais il est éventuellement possible d'accèder indépendamment à chaque colonne par son nom
- **ResultSetMetaData**: Un objet contenant des informations sur le nom et le type des colonnes d'une table

A chacun de ces objets est associé un grand nombre de méthodes permettant d'obtenir des informations très détaillées sur les éléments de la base de données.

Toutefois, seul un nombre réduit des méthodes de chacun de ces objets permet d'obtenir les informations les plus importantes sur les données.

L'objet *ResultSet* est l'objet le plus important de la technologie JDBC, car il s'agit d'une abstraction de la table ou de la réponse à une requête (généralement un sous-ensemble d'une table). Ainsi, presque toutes les méthodes et requêtes retournent les données sous forme d'un objet *ResultSet*. Cet objet contient un nombre donné de colonnes repérées chacune par un nom, ainsi qu'une ou plusieurs lignes contenant les données, et auxquelles il est possible d'accèder séquentiellement une à une du haut vers le bas. Ainsi, afin d'exploiter un objet *ResultSet*, il est nécessaire de récupérer le nombre de colonnes de celui-ci, à l'aide de l'objet *ResultSetMetaData*.

```
ResultSetMetaData rsmd;

rsmd = results.getMetaData();

numcols = rsmd.getColumnCount();
```

Lors de l'obtention d'un objet *ResultSet*, le descripteur pointe avant la première ligne.

La méthode *next()* permet d'obtenir chacune des lignes suivantes, et retourne *false* lorsqu'il ne reste plus aucune ligne. Etant donné que l'extraction de données de la base peut générer des erreurs, il est indispensable d'inclure ces manipulations dans un bloc d'exception (*try*).

Les données contenues dans le *ResultSet* peuvent être obtenues sous différentes formes selon le type de données stockées dans chaque colonne. Le contenu de chaque colonne peut être obtenu soit par le nom de celle-ci, soit par son numéro (sachant que les numéros de colonne commencent à 1 et non à 0).

Les principales méthodes de l'objet *ResultSet* sont les suivantes :

- *getInt(int)*: récupère sous forme d'entier le contenu d'une colonne désignée par son numéro
- *getInt(String)*: récupère sous forme d'entier le contenu d'une colonne désignée par son nom

- *getFloat(int)*: récupère sous forme de flottant le contenu d'une colonne désignée par son numéro
- *getFloat(String)*: récupère sous forme de flottant le contenu d'une colonne désignée par son nom
- *next()*: déplace le pointeur de ligne sur la ligne suivante
- *close()*: "ferme" l'objet
- *getMetaData()*: retourne les métadonnées de l'objet (l'objet *ResultSetMetaData*)

L'objet *ResultSetMetaData* (obtenu de l'objet *ResultSet*) permet de connaître le nombre, le nom et le type de chaque colonne à l'aide des méthodes suivantes :

- *getColumnCount()*: récupère le nombre de colonnes
- *getColumnName(int)*: récupère le nom de la colonne spécifiée
- *getColumnLabel(int)*: récupère le label de la colonne spécifiée
- *getColumnType(int)*: récupère le type de données de la colonne spécifiée

L'objet *DataBaseMetaData* permet d'obtenir des informations sur la base de données entière. Il sert notamment à récupérer le nom des tables contenues dans la base de données. De plus, étant donné que de nombreuses bases de données supportent des variantes du langage SQL, il existe de nombreuses méthodes associées à l'objet *DataBaseMetaData* permettant de connaître les méthodes SQL supportées par la base.

Création d'objets JDBC de plus haut niveau

Puisque l'accès à une base de données nécessite l'utilisation conjointe de plusieurs objets, il peut être intéressant de créer quelques objets de plus haut niveau encapsulant la plupart des comportements cités ci-dessus.

Ainsi la création d'un objet *DataBase* pour permettre d'encapsuler l'ensemble des objets nécessaires à la connexion à une base de données (*Connection*, *Statement*, *DataBaseMetaData*), ainsi que de (re)définir des méthodes simples permettant de rendre plus simples certaines opérations, comme la création de la connexion, la récupération du nom des tables, ainsi qu'une méthode *Execute* rendant l'exécution de requête triviale.

```
class Database
{

Connection con;

resultSet results;

ResultSetMetaData rsmd;

DatabaseMetaData dm;

String catalog;

String types[];
```

```
//----------------------------
public Database(String Driver)
{

 types = new String[1];

 types[0] = "TABLES";

 try {
 Class.forName(driver);

 }
 catch(Exception e){
  System.out.println("Erreur lors du chargement du driver:"+ e.getMessage());

 }
}

//-------------------------------------------------------
public void Open(String url,String login,String password)
{

 try{
  con = DriverManager.getConnection(url,login,password);

  dma = con.getMetaData();

  results = new resultSet(dma.getCatalogs());

  String s[];

  while(results.hasMoreElements()) {
   s = results.NetxElement();

  }
 }
 catch(Exception e){
  System.out.println("echec d'ouverture:"+e.getMessage());

 }
}

//-----------------
public void Close()
{

 try{
  con.close();

 }
 catch(Exception e){
  System.out.println("echec lors de la fermeture:"+e.getMessage());

 }
}

//----------------------------
```

```
public String[] getTableNames()
{

 String[] tbnames = null;

 Vector tname = new Vector();

 try{
  results = new resultSet(dma.getTables(catalog,null,"%",types));

  while (results.hasMoreElements())
   tname.addElement(results.getColumnValue("TABLE_NAME"));
 }
 catch(Exception e){
  System.out.println(e.getMessage());

 }
 tbnames = new String[tname.size()];

 for(int i=0;i<tname.size();i++)
  tbnames[i] = (String)tname.elementAt(i);

 return tbnames;

}

//---------------------------------
public String[] getTableMetaData()
{

 results = null;

 try{
  results = new resultSet(dma.getTables(catalog,null,"%",types));

 }
 catch(Exception e){
  System.out.println(e.getMessage());

 }
 return results.getMetaData();

}

//-------------------------------------------------
public String[] getColumnMetaData(String tablename)
{

 results = null;

 try{
  results = new resultSet(dma.getTables(catalog,null,tablename,null));

 }
 catch(Exception e){
  System.out.println(e.getMessage());
```

```
 }
 return results.getMetaData();

}

//-------------------------------------------------
public String[] getColumnNames(String table)
{

 String[] tbnames = null;

 Vector tname = new Vector();

 try{
  results = new resultSet(dma.getTables(catalog,null,table,null));

  while (results.hasMoreElements())
   tname.addElement(results.getColumnValue("COLUMN_NAME"));
 }
 catch(Exception e){
  System.out.println(e.getMessage());

 }
 tbnames = new String[tname.size()];

 for(int i=0;i<tname.size();i++)
  tbnames[i] = (String)tname.elementAt(i);

 return tbnames;

}

//-------------------------------------------------
public void getColumnValue(String table, String columnName)
{

 try{
  if (table.length()>0)
   results = Execute("Select "+columnName+" from "+table+" order by
"+columnName);

 }
 catch(Exception e){
  System.out.println("Erreur sur la valeur de la colonne "
                                        +columnName+e.getMessage());

 }
}

//-------------------------------------------------
public String getNextValue(String columnName)
{

 String res = "";

 try{
  if (results.hasMoreElements())
```

```
   res = results.getColumnvalue(columnName);

 }
 catch(Exception e){
  System.out.println("Erreur sur la valeur suivante
"+columnName+e.getMessage());

 }
 return res;

}

//-------------------------------------------------
public resultSet Execute(String sql)
{

 results = null;

 try{
  Statement stmt = con.createStatement();

  results = new resultSet(stmt.executeQuery(sql));

 }
 catch(Exception e){
  System.out.println(e.getMessage());

 }
 return results;

}

}
```

De la même façon, un nouvel objet *resultSet* (avec un *r* minuscule) peut être intéressant s'il permet de façon transparente de retourner automatiquement les résultats sous forme d'un tableau de valeurs, ainsi qu'en encapsulant le nombre et le nom des colonnes contenues dans l'objet *resultSet*...