*Jarboui Yassine*

# Introduction à la programmation Objet

# Le langage JAVA

**Année 2000**

# TABLE DES MATIERES


**1. INTRODUCTION ..... 1**

1.1 BIBLIOGRAPHIE ..... 1
1.2 ORIGINES DU LANGAGE ..... 1
1.3 LES POINTS FORTS DE JAVA ..... 1
1.4 LES POINTS FAIBLES ..... 3
1.5 LA PLATE-FORME DE DEVELOPPEMENT JAVA ..... 3
1.6 LE PROGRAMME « HELLO WORLD » ..... 4

**2. VOUS AVEZ DIT OBJET ? ..... 5**

2.1 INTRODUCTION ..... 5
2.2 NOTION D'OBJET ..... 6
2.3 ENCAPSULATION, METHODES ET INVOCATION ..... 7
2.4 NIVEAUX D'ABSTRACTION ET CLASSES D'OBJETS ..... 8
2.5 LE POLYMORPHISME ..... 8
2.6 POURQUOI UTILISER L'APPROCHE OBJET ? ..... 9

**3. ELEMENTS DE BASE (== C/C++) ..... 10**

3.1 LES TYPES DE BASE ..... 10
3.2 LA DECLARATION DES VARIABLES ET LES CONSTANTES ..... 10
3.3 LES OPERATEURS ..... 11
3.4 CONVERSION DE TYPE ..... 11
3.5 LES STRUCTURES DE CONTROLE ..... 12
*3.5.1 Les principales différences avec le C ..... 12*
*3.5.2 if ..... 12*
*3.5.3 switch ..... 12*
*3.5.4 while ..... 13*
*3.5.5 for ..... 13*
*3.5.6 break et continue ..... 14*
3.6 LES COMMENTAIRES ..... 14

**4. OBJETS, CLASSES ..... 16**

4.1 LES CLASSES ..... 16
*4.1.1 Définition, le mot clé class ..... 16*
*4.1.2 Modificateur de classe : public ..... 16*
*4.1.3 Modificateur de classe : final ..... 17*
4.2 LES ATTRIBUTS (== LES CHAMPS) ..... 17
*4.2.1 Déclaration et initialisation ..... 17*
*4.2.2 Le mot clé : final ..... 18*
*4.2.3 Le mot clé : static ..... 18*
4.3 LES CONVENTIONS DE NOMMAGE ..... 19
4.4 LES METHODES ..... 20
*4.4.1 Le passage des paramètres ..... 20*
4.4.1.1 Les types de bases : passage de paramètres par valeur ..... 20
4.4.1.2 Les objets : passage de paramètres par adresse ..... 20
*4.4.2 La surcharge des méthodes ..... 21*
*4.4.3 Le mot clé : static ..... 22*
*4.4.4 Les constructeurs ..... 23*
*4.4.5 Le bloc d'initialisation ..... 24*

*4.4.6 La méthode de finalisation : finalize()* ..... *24*
4.5 LES REFERENCES ..... 25
*4.5.1 La création d'objet* ..... *25*
*4.5.2 La référence sur l'objet courant : this* ..... *26*
*4.5.3 La référence vide : null* ..... *26*
*4.5.4 Les opérateurs sur les références* ..... *27*
4.6 LES TABLEAUX ..... 27

**5. L'HERITAGE ..... 29**

5.1 PRINCIPES GENERAUX, LE MOT CLE EXTENDS ..... 29
5.2 LE POLYMORPHISME ..... 31
5.3 ACCES A LA SUPER-CLASSE D'UNE CLASSE : SUPER(…) ..... 32
5.4 METHODES ET CLASSES ABSTRAITES : ABSTRACT ..... 33

**6. LES PACKAGES ET L'ENCAPSULATION ..... 35**

6.1 LES PACKAGES ..... 35
6.2 LA STRUCTURE DE STOCKAGE DES CLASSES ET DES PACKAGES ..... 36
6.3 LES PRINCIPAUX PACKAGES DU JDK ..... 37
6.4 LES REGLES DE VISIBILITE DES ATTRIBUTS ET METHODES : PUBLIC, PROTECTED, PRIVATE, FRIENDLY 37
6.5 LES CLASSES IMBRIQUEES ET LES CLASSES ANONYMES ..... 38

**7. LES INTERFACES ..... 40**

7.1 DEFINITIONS, LES MOTS CLES INTERFACE ET IMPLEMENTS. ..... 40
7.2 INTERFACES ET HERITAGE ..... 41
7.3 LES REFERENCES DE TYPE INTERFACE, L'OPERATEUR INSTANCEOF. ..... 41

**8. LES EXCEPTIONS ..... 43**

8.1 POURQUOI DES EXCEPTIONS ? ..... 43
8.2 REPRESENTATION ET DECLENCHEMENT DES EXCEPTIONS : CLASSE EXCEPTION, OPERATEUR THROW 43
8.3 PROPAGATION, CAPTURE ET TRAITEMENT DES EXCEPTIONS ..... 44
*8.3.1 Propagation des exceptions : le mot clé throws* ..... *44*
*8.3.2 Capture des exceptions : catch, try et finally* ..... *45*

**9. LES CLASSES DE BASE ..... 48**

9.1 LA CLASSE OBJECT ..... 48
9.2 LES CLASSES WRAPPER ..... 48
9.3 LES CHAINES DE CARACTERES ..... 49
*9.3.1 La classe java.lang.String* ..... *50*
*9.3.2 La classe java.lang.StringBuffer* ..... *50*
*9.3.3 La classe java.util.StringTokenizer* ..... *51*
9.4 LES CONTENEURS ..... 51
*9.4.1 La classe java.util.Vector* ..... *52*
*9.4.2 L'interface java.util.Enumeration* ..... *52*
*9.4.3 La classe java.util.Hashtable* ..... *53*

**10. L'INTERFACE HOMME/MACHINE : AWT ..... 55**

10.1 INTRODUCTION ..... 55
10.2 LES COMPOSANTS : JAVA.AWT.COMPONENT ..... 55
10.3 LES CONTENEURS : JAVA.AWT.CONTAINER ..... 57
10.4 LES GESTIONNAIRES D'ASPECT : JAVA.AWT.LAYOUTMANAGER ..... 58
*10.4.1 Généralités* ..... *58*
*10.4.2 Java.awt.FlowLayout* ..... *59*
*10.4.3 Java.awt.BorderLayout* ..... *60*
10.5 LA GESTION DES GRAPHIQUES : JAVA.AWT.GRAPHICS ..... 61
10.6 LA GESTION DES EVENEMENTS ..... 62
*10.6.1 Principes* ..... *62*
*10.6.2 Les sources d'événements* ..... *62*

*10.6.3 Les événements de l'AWT* ..... *63*
*10.6.4 Listener d'événements* ..... *63*

**11. LES ENTREES/SORTIES ..... 65**

11.1 GENERALITES ..... 65
11.2 LA CLASSE JAVA.IO.FILE ..... 66
11.3 EXEMPLES D'ENTREES/SORTIES PAR OCTET ..... 66
*11.3.1 Utilisation des canaux FileInputStream et FileOutputStream* ..... *66*
*11.3.2 Utilisation des filtres BufferedInputStream et BufferedOutputStream* ..... *67*
*11.3.3 Utilisation des Filtres DataInputStream et DataOutputStream* ..... *68*
11.4 EXEMPLES D'ENTREES/SORTIES PAR CARACTERE ..... 69
*11.4.1 Utilisation des canaux FileReader et FileWriter* ..... *69*
*11.4.2 Utilisation des filtres BufferedReader et BufferedWriter* ..... *69*
*11.4.3 Utilisation du filtre PrintWriter* ..... *69*
*11.4.4 Utilisation de OutputStreamWriter et InputStreamReader* ..... *69*

**12. LES THREADS ..... 71**

12.1 PRINCIPES GENERAUX ..... 71
12.2 MISE EN œUVRE AVEC LA CLASSE JAVA.LANG.THREAD ..... 71
12.3 MISE EN œUVRE AVEC L 'INTERFACE JAVA.LANG.RUNNABLE ..... 73
12.4 PARTAGE DU TEMPS ..... 73
12.5 PARTAGE DE DONNEES ..... 74

**13. LE RESEAU ..... 76**

13.1 LES URL ..... 76
*13.1.1 Définitions* ..... *76*
*13.1.2 Les classes URL et URLConnection* ..... *76*
13.2 LES SOCKETS ..... 77
*13.2.1 Présentation* ..... *77*
*13.2.2 Le client : classes InetAddress et Socket* ..... *77*
*13.2.3 Le serveur : classes ServerSocket et Socket* ..... *78*

**14. LES APPLETS ..... 81**

14.1 APPLETS ET WEB ..... 81
14.2 LA CLASSE JAVA.APPLET.APPLET ..... 81
14.3 METHODES UTILES DE LA CLASSE APPLET ..... 82
14.4 APPLETS ET SECURITE ..... 83
14.5 APPLETS ET THREADS ..... 83
14.6 L'INTERFACE APPLETCONTEXT ET LA METHODE GETAPPLETCONTEXT() ..... 85
*14.6.1 Présentation* ..... *85*
*14.6.2 Exemple : Interactions avec le navigateur* ..... *86*
*14.6.3 Exemple : Communication inter-applets* ..... *86*
*14.6.4 Exemple : Chargement d'images* ..... *87*

# 1. Introduction

Ce cours est une introduction à la programmation objet et est basé sur l'apprentissage du langage Java. Il suppose de connaître le langage C ou C++.

## 1.1 Bibliographie

Java est un langage en rapide évolution, il est nécessaire avant tout de consulter les livres les plus récents ou de se connecter à Internet !

Quelques livres :

- Java par la pratique de Pat Niemeyer et Josh Peck. *Editions O'Reilly*
- Java client-serveur de Cédric Nicolas, Christophe Avare Frédéric Najman. *Editions Eyrolles*
- Programmation réseau avec Java de Elliotte Rusty Harold. *Editions O'Reilly*

Quelques liens :

- http://www.javasoft.com/
- http://www.sunsoft.com/
- http://www.javaworld.com/

## 1.2 Origines du langage

Java a été développé par SunSoft (branche de Sun MicroSystems) pour réaliser le développement de systèmes embarqués.

Dés le début, des versions gratuites du compilateur et les spécifications du langage étaient disponibles sur Internet : D'où un développement rapide et consensuel du langage.

Aujourd'hui les évolutions de Java sont gérées par JavaSoft, dépendant de SunSoft, avec le partenariat de nombreuses grandes entreprises (Borland / Inprise, IBM, …).

Le langage est actuellement utilisé dans de nombreux domaines de l'informatique : Réseaux /Internet (qui l'a rendu célèbre), développement rapide d'application (RAD), applications clients/serveurs, systèmes embarqués (Voitures, TV, Téléphones portables,… machines à laver),…

Java est un langage qui s'inspire de la syntaxe de **C++** et de la gestion dynamique de la mémoire de **SmallTalk**, deux langages orientés objets. On s'accorde à dire que java s'inspire à 70% de C++, et à 30% de langages comme SmallTalk, ADA et Objective C.

## 1.3 Les points forts de JAVA

- ***Java est simple*** par rapport à C et C++, les sources de nombreux bugs ont été supprimées:
  - Pas de pointeurs
  - Pas de surcharge des opérateurs
  - Pas d'héritage multiple
  - Développements 30 à 35% plus rapides

- ***Java est un langage Objet*** :
  - Syntaxe de C++, gestion dynamique de la mémoire de SmallTalk

- Langage objet pur, contrairement à C++, qui permet une programmation de type C
- Compromis entre un langage objet statique (C++) et dynamique (SmallTalk)
- Livré avec des packages de classes riches et variés (Réseau, Web, Base de Donnée, Téléphonie, …)

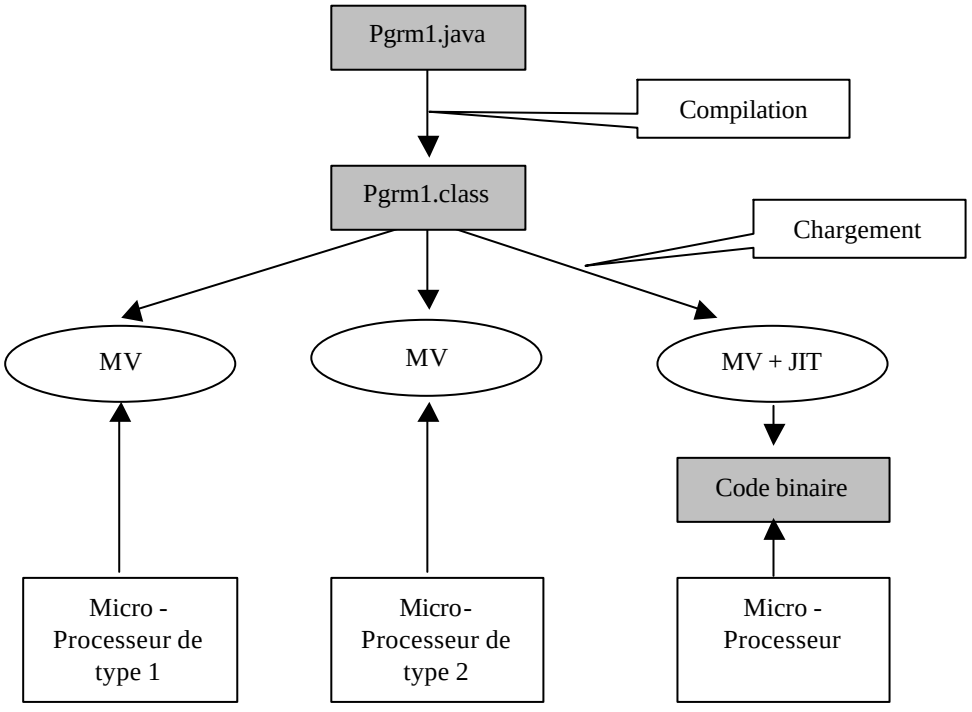
➢ ***Java est robuste*** :

- Compilateur très strict, en particulier car :
  - ♦ Contrôle de typage fort (pertes de précision interdites)
  - ♦ Vérification que les erreurs identifiées (appelées exception) sont traitées par le développeur
- La gestion de la mémoire n'est plus à la charge du développeur (Garbage collector, pas de pointeurs)
- Un débordement d'index dans un tableau provoque une erreur ( => la mémoire n'est pas écrasée)

➢ ***Java est portable*** :

- Un code source Java est compilé pour générer un code assembleur virtuel : Le byte code.
- Le byte code est exécuté par un logiciel : la **Machine Virtuel**. Seule la Machine Virtuelle change d'une machine à l'autre. Un programme Java est donc exécutable sur toute machine pour laquelle une machine virtuelle existe.
- Le byte code est conçu pour :
  - ♦ Etre rapidement interprété par une machine virtuelle (java est en fait un langage semi-compilé et semi-interprété)
  - ♦ Etre optimisable au moment de son exécution (dépendant de la plate-forme d'exécution)
  - ♦ Etre compact (en moyenne une instruction byte code = 1,8 octets, RISC = 4 octets)
  - ♦ Etre vérifiable (système de vérification d'intégrité du code)

❖ La plupart des navigateurs Web intègrent une machine virtuelle Java, leur permettant de charger des petites applications Java avec les pages HTML (les **Applets, parfois traduites en Applicules**). Si la notoriété de Java doit beaucoup aux Applets, celles-ci ne constituent qu'une petite partie des domaines d'utilisation possibles de ce langage.



- ***Java est performant*** :
  - Des techniques récentes d'optimisation et de compilation à la volée du byte code, intégrés à certaines machines virtuelles font que ses performances approchent celles de langages compilés.

- ***Java est sécurisé*** :
  - Le **compilateur** interdit toute manipulation directe de la mémoire
  - Le **Verifier**, utilisé lors du chargement du code vérifie que le byte code est intègre et bien conforme aux règles de compilation (empêche les trucages du code) et que les méthodes référencées sont disponibles.
  - Le **ClassLoader**, responsable du chargement des classes dans la machine virtuelle, vérifie que la classe chargée n'écrase pas une autre classe dans l'environnement alloué à l'application (appelé SandBox). Dans le cas d'applets chargées à partir de serveurs Web (http), une SandBox différente est allouée pour chaque serveur Web => Il est impossible à deux applets chargées à partir de serveurs différents de communiquer entre elles (ce qui n'est pas le cas si elles sont chargées à partir du même serveur).
  - Le **SecurityManager**, actif lors de l'exécution du byte code permet de limiter les droits d'accès de l'application Java ; Par exemple, une Applet chargée à partir d'un serveur Web distant ne pourra pas accéder au système de fichier, d'utiliser une DLL, ou d'ouvrir une connexion réseau autre que vers le serveur d'origine de l'Applet.

  - ❖ Il est possible de développer un Verifier, un ClassLoader ou un SecurityManager spécifique (Ceux-ci étant fournis sous la forme de classes avec le langage).

- ***Java supporte le multi-threading*** (un Thread est une tache légère qui partage le même espace de travail que les autres Threads)
  - Intégré dans le langage de manière simple
  - Plus performant et portable que l'utilisation d'un système multitâche (du type du **fork** Unix)
  - Basé sur des mécanismes de multi-threading du système d'exploitation hôte, lors de l'exécution (=> performances).

## 1.4 Les points faibles

Ce sont les conséquences des points forts que nous venons d'énumérer, en particulier :

- Le développeur n'a aucune visibilité sur ce qui se passe en mémoire.

- Java est encore actuellement moins performant qu'un bon langage compilé (C, C++)

## 1.5 La plate-forme de développement JAVA

Le **JDK** (Java Development Kit) est l'outil de base pour tout développement Java. Il est gratuit ! (il peut être chargé sur le site JavaSoft).

Ce kit contient tout le nécessaire pour développer des applications ou des applets Java : Le compilateur (en ligne de commandes), une machine virtuelle, un ensemble complet de classes de bases regroupées en packages.

Le langage est toujours en évolution rapide : La version actuelle est la 1.1 (après la 1.02 et avant la 1.2).

Différents outils du commerce fournissent des interfaces graphiques permettant d'encapsuler le JDK. Une partie des développements peut alors être réalisée de manière visuelle. Nous utiliserons en TP un de ces outils : Jbuilder 2.0 de la société Inprise (Borland).

Les classes de bases (livrées avec le JDK) concernent :

- L'interface homme /machine (AWT : Abstract Windows Toolkit) : fenêtres, boutons, listes,…
- Le Réseau : TCP/IP, Web, Applet, RMI (appels distants)
- Les entrées /sorties : accès direct, fichiers,…
- Des extensions du langage : Conteneurs, dates, chaînes de caractères,…
- L'accès aux bases de données : JDBC
- La possibilité de créer des composants Java : Les Beans
- La gestion de la sécurité : Cryptage, identification, signature

D'autres Packages livrés séparément concernent :

- Le multimédia (2D, 3D, Son, …)
- La téléphonie
- La manipulation de mail
- …

## 1.6 Le programme « Hello World »

Avant d'aller plus en avant dans ce cours, voici l'exemple le plus simple de programmation Java :

```
class Hello {

        public static void main(String argv[])
        {
                System.out.println("Hello World");
        }
}
```

Les étapes nécessaires pour développer ce programme et l'exécuter dans une fenêtre DOS sans outils de développement visuel (uniquement avec le JDK) sont:

```
C:> EDIT Hello.java          <= Edition
C:> javac Hello.java         <= Compilation : création de Hello.class
C:> java Hello               <= Lancement du programme à partir de la machine virtuelle
Hello World
C:>
```

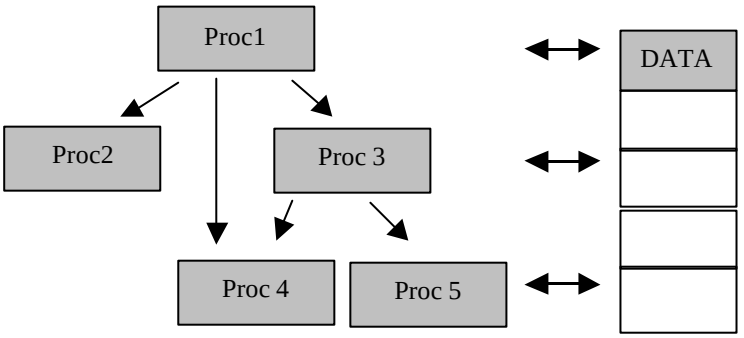
# 2. Vous avez dit Objet ?

## 2.1 Introduction

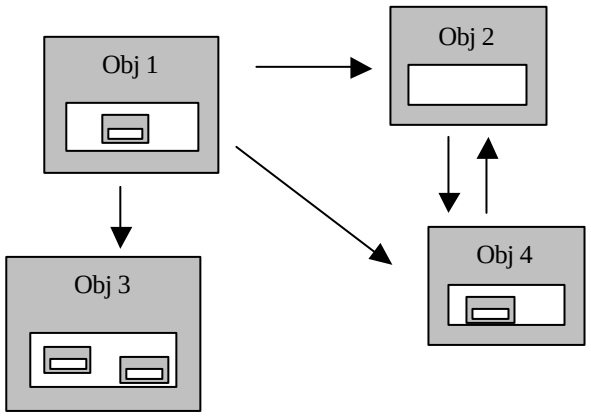
Java est un langage orienté Objet. Il est donc nécessaire de bien comprendre ce qu'implique cette notion avant d'entreprendre l'étude de ce langage.

L'approche Objet est, en programmation, une approche fondamentalement différente de l'approche procédurale que vous avez pu aborder avec un langage tel que le C :

- Dans une approche procédurale, il y a séparation complète des notions de Données et d'Actions (de code) : le développeur part des données (de quoi dispose-t'on, que veux t'on obtenir), pour ensuite définir la série des actions qui permettrons de transformer les données initiales en résultats souhaités. Ces actions sont décomposées de manière hiérarchique (une action d'un certain niveau est la composition d'actions du niveau inférieur). Ce qui permet d'obtenir un certain niveau d'organisation et de réutilisation (les procédures ou fonctions).




- Dans l'approche Objet, il n'y a plus séparation des données et des actions : Le travail du développeur consiste à faire interagir des objets qui contiennent chacun les données permettant de représenter un élément logique de programmation, ainsi que les méthodes qui constituent le seul moyen d'agir sur ces données. Dans un programme objet, on dit qu'il y a coopération entre les objets qui le composent : chaque objet peut invoquer une méthode d'un autre objet qui coopère en répondant à cette demande. Chaque objet est lui-même la composition d'un ensemble d'objets coopératifs.



L'approche objet est en fait une manière totalement différente d'aborder un problème. La philosophie sous-jacente est beaucoup plus générale qu'une simple méthode de programmation : elle se rapporte à la manière que nous avons de voir et structurer le monde qui nous entoure, de le modéliser, ce qui nous permet d'organiser notre pensée pour résoudre nos problèmes quotidiens.

## 2.2 Notion d'objet

C'est une notion très proche de celle de la vie courante : Nous nous représentons en permanence le monde qui nous entoure sous la forme d'un ensemble d'objets plus ou moins abstraits, avec lesquels nous interagissons et qui interagissent entre eux: des tables, des chaises, des stylos, des personnes, des animaux...

La vision que nous avons d'un certain objet peut être différente en fonction de nos besoins, c'est à dire en fonction du problème que l'on a à résoudre, du mécanisme que l'on doit décrire, plus littéralement « de notre point de vue » :

- Un policier dont le rôle est de régler la circulation voit un ensemble de véhicules, objets interagissant entre eux, qui doivent respecter des règles de circulation et qu'il peut à volonté faire stopper ou faire circuler. De son point de vue, un camion ou une voiture sont des objets de même nature.

- Le conducteur d'un de ces véhicules, le voit d'un autre œil : Il conduit une voiture bien particulière, et il dispose d'un ensemble complexe de commandes pour la faire avancer, la diriger, interagir avec les autres véhicules. Son problème à lui est de se déplacer d'un point à un autre, si possible rapidement.

- Un garagiste, connaît les différentes pièces qui composent un véhicule, ainsi que son fonctionnement interne : pour lui un véhicule est lui-même un ensemble d'objets interagissant entre eux.

- Le concepteur du véhicule, intervient à un niveau de décomposition encore plus bas : Il peut par exemple se pencher sur la structure atomique de tel matériau utilisé dans la construction de son véhicule pour des raisons de solidité de celui-ci. (un atome est lui-même une abstraction de la réalité : noyau, électron, …)

On voit à travers cet exemple, que parler de l'objet véhicule représente un certain niveau d'abstraction : on fait abstraction de la réalité (mais qu'est-ce que la réalité ?) pour s'intéresser à un niveau de représentation de cet objet suffisant pour nous permettre de résoudre notre problème : il peut être intéressant de descendre davantage dans le détail (la voiture, les pièces composant une voiture, …), mais on ne le fera que si cela est nécessaire (si notre problème change et par conséquent on doit changer de niveau d'abstraction).

- Le policier et le conducteur de l'exemple précédent n'ont pas du tout la même vision du même objet.

Un objet est défini par la manière dont on peut interagir avec lui et cela en fonction du problème que l'on a à résoudre. Il représente un tout logique du point de vue du problème qui nous intéresse : peu importe ses composants et la manière dont il fonctionne exactement, peu importe s'il peut faire autre chose si cela ne nous est pas utile. Il correspond à un certain niveau d'abstraction de la réalité et est représenté essentiellement par les services qu'il peut nous rendre (que nous pouvons lui demander).

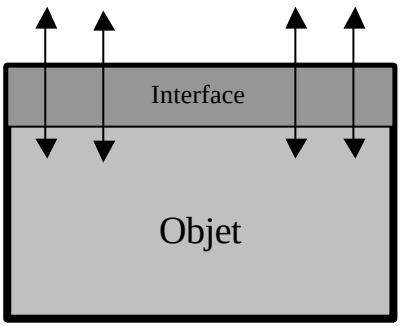
Nous faisons tous en permanence un tel travail d'abstraction pour pouvoir voir et comprendre simplement le monde qui nous entoure.

## 2.3 Encapsulation, méthodes et invocation

Du point de vue de celui qui l'utilise, un objet est une boite noire qui offre un certain nombre de fonctions ou **méthodes** permettant d'interagir avec lui. Peu importe comment il est construit de manière interne, la seule chose nécessaire pour pouvoir utiliser un objet est savoir ce qu'il peut faire et surtout, comment lui demander :

- Un poste de Télévision moderne est une boite ayant pour interface : un écran, des haut-parleurs et une télécommande. Pour changer de chaîne il suffit de demander à cette boite noire de le faire pour nous, en appuyant simplement sur le bouton correspondant. Peu importe ce qui se passe réellement dans le poste.

L'ensemble des **méthodes** proposées par un objet est appelé **l'interface** de cet objet.
On dit qu'un objet est **encapsulé** par son **interface :** la seule manière d'interagir avec cet objet est **d'invoquer** une des méthodes de son interface. Peut importe de quoi cet objet est réellement constitué, ce qui est important c'est les services (les méthodes) qu'il peut fournir.




**L'encapsulation** d'un objet par son interface permet de masquer son contenu et donc offre la possibilité de modifier celui-ci sans aucun impact pour le reste du monde. Respecter ce principe d'encapsulation, nous permet aussi de pouvoir utiliser immédiatement des objets de même nature, même si leur fonctionnement interne est différent.

Cela constitue un des points centraux de l'approche objet : Pour respecter ce principe d'encapsulation on ne doit interagir avec un objet que par l'invocation d'une des méthodes de son interface. En faisant cela **on dit à l'objet ce qu'il doit faire, jamais comment il doit le faire** (bref, on se mêle de ses oignons).

- Lorsque l'on freine avec sa voiture on se contente d'appuyer sur la pédale de frein (on invoque la méthode de freinage), on ne s'occupe pas de savoir s'il s'agit d'un frein à disque ou de la manière dont cette commande est transmise

- Un policier qui règle la circulation, invoque la méthode « arrêtez » à un véhicule en levant la main d'une certaine manière. Il ne s'occupe pas de la manière dont chaque véhicule s'arrête (ce qui dépend du type de véhicule, du type de conducteur,…)

On voit que penser objet, c'est aussi « savoir déléguer », ne pas faire le travail que d'autres ferons mieux que vous.

## 2.4 Niveaux d'abstraction et Classes d'objets

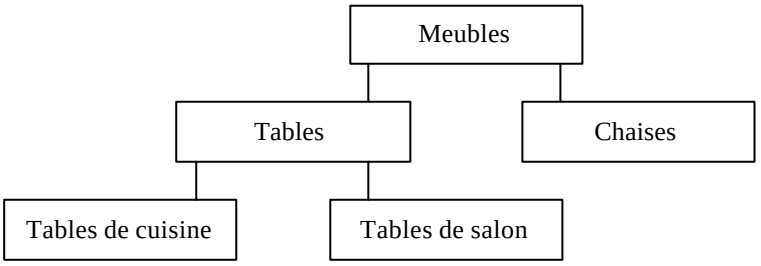
Plusieurs niveaux d'abstractions sont possibles lorsque l'on représente un certain objet « réel » : Lorsque l'on regarde la table qui est dans notre cuisine, on fait généralement abstraction de tout ce qui compose physiquement cet objet (pieds de fer, plateau, tiroir,…) pour ne voir que l'objet de notre cuisine permettant de ranger les couverts et de servir de support pour nos repas. En fait, on range alors implicitement cette table dans l'ensemble général des tables de cuisine. Cet ensemble regroupe tous les objets ayant les mêmes fonctionnalités que notre table. En langage objet nous parlerons de la **Classe** des tables de cuisines et nous dirons que notre table de cuisine particulière est une **instance** de cette classe (C'est à dire un représentant de cette classe).

Une classe est définie en décrivant l'interface commune à tous les objets qui la composent (qui sont des instances de celle-ci).

Si l'on cherche un support pour écrire une lettre, nous chercherons de manière plus générale une table permettant de fournir le service « support ». Nous pourrons choisir la table de la cuisine, du salon ou de la table à manger. La classe des tables de cuisine est donc comprise dans celle plus générale des tables (qui inclut aussi la classe des tables de salon,…). La classe des tables est un niveau d'abstraction supérieur à celle des tables de cuisine : elle représente un ensemble d'objets moins spécialisés que la classe des tables de cuisine. Une table de cuisine est une table qui possède des caractéristiques spécifiques. On dit que la classe des tables de cuisine est **une classe fille** de la classe des tables et qu'elle **hérite** des caractéristiques de cette classe, tout en la spécialisant par l'ajout de nouvelles fonctionnalités.
Si la classe des tables est déjà définie, et si l'on veut définir celle des tables de cuisine, il suffira de dire que cette nouvelle classe hérite de la précédente, avec en plus, telle ou telle nouvelle caractéristique.

Une classe fille est définie à partir de sa mère par **héritage** (réutilisation de ce qui est déjà défini) et **spécialisation** (ajout de nouvelles fonctionnalités).

En fonction de nos besoins et du moment, nous pourrons considérer notre table, soit comme notre table de cuisine (celle qui a un pied bancal), soit comme une table de cuisine (là où nous voulons mettre la table), soit comme une table quelconque (pour écrire une lettre).




Cette classification en groupes de plus en plus généraux est une démarche que nous accomplissons naturellement : Une 205 Peugeot est une instance de la classe des 205 Peugeot qui est incluse dans celle des voitures Peugeot, incluse dans celle des voitures, incluse dans celle des véhicules,…En fonction des circonstances, nous la considérerons d'une manière plus ou moins précise.

De la même façon Albert fait partie de la classe des hommes, inclue dans celle des êtres humains, qui fait partie des omnivores, donc des mammifères et de manière plus générale des êtres vivants.

## 2.5 Le polymorphisme

Le **polymorphisme** (du grec « plusieurs formes »), signifie que la même méthode peut être définie dans différentes classes et que chaque classe peut implémenter (réaliser) cette méthode à sa manière. Il est ainsi parfaitement possible d'invoquer une méthode sur un objet sans savoir à quelle classe celui-ci appartient exactement et donc sans savoir quelles seront exactement les actions réalisées :

- ❖ Lorsque l'on freine dans une voiture on sait que la fonction attendue est de stopper le véhicule. En fonction du type réel de véhicule (de sa classe plus spécifique) les actions réelles seront très différentes (ABS ou non, freins à tambour, électromagnétiques,…). Le conducteur n'a pas besoin de connaître ce niveau de détail pour savoir s'arrêter.

- ❖ Un policier qui règle la circulation invoque toujours la même méthode pour arrêter des objets de la classe des véhicules. Chaque véhicule, en fonction de sa classe spécifique utilisera cependant une méthode particulière pour s'arrêter : un cycliste ne s'arrêtera pas comme une voiture ou un camion…

## 2.6 Pourquoi utiliser l'approche objet ?

La motivation essentielle de cette approche est d'augmenter les possibilités de réutilisation de ce que nous développons : Le monde de la programmation est encore proche du stade artisanal où chaque programmeur passe son temps à redévelopper ce que d'autres (souvent lui-même) ont déjà réalisé.

Les moyens de réutilisation offerts par l'approche objet sont multiples :

- L'encapsulation des données et du code dans une même entité permet de garantir la cohérence des objets (qui se suffisent à eux-mêmes, contrairement aux procédures qui dépendent la plupart du temps de données externes à elles-mêmes). Cette cohérence est indispensable pour envisager de réutiliser un objet dans un autre contexte (pas suffisant, mais nécessaire)

- La notion d'encapsulation par une interface permet de normaliser le fonctionnement des objets : il est possible de changer le fonctionnement interne d'un objet particulier, sans modifier la manière de l'utiliser (c'est à dire le reste du programme)

- La notion d'héritage permet de réutiliser ce qui a déjà été défini lors de la conception d'un objet pour en créer de nouveaux.

- La notion de polymorphisme, permet de manipuler de manière identique des objets ayant des comportements totalement différents (des comportements qui ne sont pas obligatoirement connus au moment où l'on définit ces manipulations).

De manière plus générale, l'approche objet permet de construire des programmes mieux conçus, car plus évolutifs et souvent plus sûrs.

# 3. Eléments de Base (== C/C++)

La plupart de ces éléments sont des reprises directes du langage C (et C++).

Nous les passerons donc vite en revue, en ne nous attardant que sur les différences entre Java et C.

## 3.1 Les types de Base

Java dispose de types de base comme le langage C. Ces types doivent être considérés comme des classes d'objets prédéfinies et optimisées. Ils constituent malgré tout une entorse à la pureté du modèle objet de Java.

Les types de base sont :

| | ***Taille*** | ***Valeurs*** | ***Normes*** | ***Valeur inférieure*** | ***Valeur supérieure*** |
|---|---|---|---|---|---|
| **byte** | 1 octet | signé | Format IEEE | -256 | 255 |
| **short** | 2 octets | signé | Format IEEE | -32768 | 32767 |
| **int** | 4 octets | signé | Format IEEE | -2147483648 | 2147483647 |
| **long** | 8 octets | signé | Format IEEE | -9223372036854775808 | 9223372036854775807 |
| **float** | 4 octets | signé | Format IEEE | | |
| **double** | 8 octets | signé | Format IEEE | | |
| **char** | 2 octets | Codage Unicode | | | |
| **boolean** | 1 bit | 2 valeurs : true ou false | | false | true |

Remarques :

- Tous les types numériques de Java sont signés
- Le type **int** fait toujours 4 octets et le type **long** en fait 8.
- Les caractères ne sont pas codés en ASCII, mais en Unicode. Le codage Unicode est une extension du codage ASCII : les 128 premiers caractères Unicode sont les mêmes que ceux de l'ASCII, et au de là de 128, de nouveaux caractères sont définis (par exemple les caractères accentués que l'on peut donc manipuler sans problème dans un programme Java).
- Il existe un **vrai** type booléen qui ne peut prendre que deux valeurs : true ou false. ***Il n'existe pas de conversion automatique du type boolean vers un autre type*** (0 ne veux plus dire faux !)

## 3.2 La déclaration des variables et les constantes

Les exemples suivants montrent que la forme des déclarations et le format des constantes sont similaires au langage C :

```
byte    b     = 67 ;
short   si    = -24000 ;
int     i     = 42000 ;
long    l1    = 600000000 ;
long    l2    = 0xA05 ;          // En hexadécimal
int     i2    = 067 ;            // en octal
```

```
float    f       = 23.456
double   d       = 23.8 e34 ;
```

```
boolean  b       = true ;
```

```
char     c1      = 'a' ;
char     c2      = 48 ;              // Unicode
int      i2      = 'a' ;
String   s       = "Bonjour \n" ;
```

**Attention** : **String** n'est pas un type de base, mais une classe. Nous reviendrons dessus plus tard. En tout état de cause, String n'a rien à voir avec un tableau ou un pointeur.

## 3.3 Les opérateurs

Ce sont les mêmes que dans le langage C :

| ***Types d'opérateurs*** | | ***Listes des opérateurs*** | ***Types concernés*** |
|---|---|---|---|
| Arithmétiques | binaire | +, -, *, /, % | Entiers et réels (sauf le modulo) |
| | unaire | -, --, ++ | |
| | étendus | +=, -=, *=, /= | |
| Binaires bit à bit | | ~, &, ^,<<, >> | Entiers |
| Logiques | | !, &&, \|\| | Booléens |
| De comparaison (le résultat est de type boolean) | | ==, !=, <, <=, >, >= | Tous les types (y compris les références d'objets) |

## 3.4 Conversion de type

Il y a 4 types de conversions possibles:

- ***Conversion explicite***
- ***Conversion implicite lors d'une affectation***
- ***Conversion implicite lors d'une promotion arithmétique*** (un opérateur s'applique toujours sur deux arguments de même type : si ce n'est pas le cas, un des deux arguments sera converti dans le type de l'autre. Le choix de l'argument à convertir se fait selon la règle "du type le plus restreint, vers le type le plus large".
- ***Conversion implicite lors d'un passage de paramètres*** (lors de l'invocation d'une méthode)

```
int      i, j ;
short    s = 2 ;
float    f = 1.2 ;

i = (int) f ;                // conversion explicite : float => int
i = s ;                      // conversion implicite lors d'une affectation : short => int

f = i / 2.0 ;                // conversion par promotion arithmétique : int => float

        // Méthode définie par : int objet.meth( int arg )
objet.meth(s) ;              // conversion lors d'un passage de paramètre : short => int
```

***Contrairement à ce qui se passe dans C, les conversions pouvant provoquer une perte de valeur ne sont pas autorisées de manière implicite dans C : cela doit être explicitement demandé :***

```
int     i, j ;
short   s = 2 ;
float   f = 1.2 ;

i = f ;             // conversion implicite : float => int !!!! ERREUR A LA COMPILATION
```

***Il n'y a pas de conversion possible (implicite ou explicite) entre un type entier et le type boolean :***

```
int i = 0 ;

if ( i ) {          // ERREUR A LA COMPILATION
…
}

if ( i!=0 ) {       // OK
…
}
```

## 3.5 Les structures de contrôle

Ce sont les mêmes que celles du C.

### 3.5.1 Les principales différences avec le C

L'instruction **goto** n'existe plus ! (Enfin…)

Par contre, les instructions **break** et **continue** peuvent être utilisées sur plusieurs niveaux d'imbrication de boucles.

Les expressions logiques sont de type **boolean**, **pas de type entier**.

### 3.5.2 if

```
class IfApp {
        public static void main(String args[]) {
                int c = 0 ;
                if ( c == 1 ) System.out.println( " Jamais imprimé " ) ;
                else System.out.println( " Ceci sera imprimé" ) ;
        }
}
```

**Remarque** : L'expression logique attendue est obligatoirement de type boolean. Ceci permet en particulier d'éviter l'erreur classique consistant à confondre l'affectation (=) avec la comparaison (==). Par exemple si l'on écrit ***if ( i = 1 )***, le compilateur détectera une erreur, car le type de l'expression logique est alors ***int***.

### 3.5.3 switch

```
class SwitchApp {
        public static void main(String args[]) {
                System.out.println( " Entrez A, B ou C : " ) ;
                char c = (char) System.in.read( )  ;
                switch ( c ) {
                        case 'a' : case 'A' :
                                System.out.println( " A " ) ;
                                break ;
                        case 'b' : case 'B' :
                                System.out.println( " B " ) ;
                                break ;
                        case 'c' : case 'C' :
                                System.out.println( " C " ) ;
                                break ;
                        default :
                                System.out.println( " Erreur " ) ;
                                break ;
                }
        }
}
```

### 3.5.4 while

```
class WhileApp {
        public static void main(String args[]) {
                int c  ;
                do {
                        System.out.println("Entrez un nombre positif <= 5") ;
                        c = LireEntier( ) ;
                } while ( c <= 0 || c > 5 ) ;

                while ( c > 0 )  System.out.println( c--  ) ;
        }
        public static int LireEntier() {   /* Nous verrons cela plus tard */  }
}
```

### 3.5.5 for

```
class ForApp {
        public static void main(String args[]) {
                int i  ;
                for ( i=0 ; i<=5 ; i++ ) {
                        System.out.println( i ) ;
                }
                for ( int j=0 ; j<=5 ; j++ ) {
                        System.out.println( j ) ;
                }
                j = 2 ;     // ERREUR : j est local au for
        }
}
```

### 3.5.6 break et continue

Comme nous l'avons déjà dit, ces deux instructions peuvent maintenant agir sur plusieurs niveaux de boucles. Il suffit pour cela d'utiliser une étiquette ( un label ) pour nommer les différents niveaux de boucles, et de préciser le nom de l'étiquette de la boucle concernée lors de l'utilisation de l'instruction **break** ou **continue**.

Etiqueter une boucle consiste à écrire un nom (celui de cette étiquette), suivi de deux points, juste avant le début cette boucle.

Rappels :

**break** : permet d'interrompre immédiatement la boucle concernée (on passe à l'instruction suivant immédiatement la fin de cette boucle

**continue** : permet de passer immédiatement à l'itération suivante de la boucle concernée.

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

                partie : while ( true ) {

                        ….
                        tour : for ( int t=0 ; t < nbNiveaux ; t++ ) {

                                …
                                choix : do {
                                        System.out.println( "Voulez-vous : " ) ;
                                        System.out.println( "  1 : Passer au tour suivant " ) ;
                                        System.out.println( "  2 : Recommencer une partie " ) ;
                                        System.out.println( "  3 : Quitter" ) ;
                                        char c = (char) System.in.read( ) ;
                                        switch ( c ) {
                                                case '1' : continue tour ;
                                                case '2' : continue partie ;
                                                case '3' : break partie ;
                                                default : System.out.println("Erreur !) ;
                                                        break ;
                                        }
                                }
                                while ( true ) ;
                        }
                }

        }
}
```

## 3.6 Les commentaires

Java supporte deux types de commentaires :

- Tout ce qui se trouve entre /* et */ est un commentaire
- Tout ce qui sur une ligne se trouve après // est un commentaire

```
class Hello {   /* ceci
est un
commentaire */

        public static void main(String argv[])
        {
                System.out.println("Hello World") ;  // autre commentaire
        }
}
```

# 4. Objets, Classes

Nous entrons dans le vif du sujet : comment sont représentés et comment utiliser les classes et les objets dans java !

***Rappel*** : Nous avons vu que dans le modèle objet une classe est définie comme un ensemble d'objet ayant une même interface.

En programmation, une classe doit être considérée comme un nouveau type de donnée qui est défini par le programmateur et qui permet de décrire l'interface et une partie de l'implémentation de l'ensemble des objets appartenant à cette classe (de la même manière que **int**, type prédéfini, permet de décrire les entiers codés sur 4 octets).

Lorsqu'une classe d'objet a été définie, il suffit de demander au compilateur de créer (on dit ***instancier***) un nouvel objet appartenant à cette classe pour que cela se réalise (de la même manière, la déclaration **int i ;** permet de créer un nouvel entier).

## 4.1 Les classes

### 4.1.1 Définition, le mot clé class

Commençons par un exemple, voici la définition d'une classe permettant de décrire des objets représentants des interrupteurs (qui peuvent être ouverts où fermés) :

```
class Interrupteur {                        // Interrupteur est le nom de la classe
        boolean open ;                      // Ceci est un attribut (une données membre de la classe)

        voidsetState (boolean newstate) {   // Ceci est une méthode
                open = newstate ;
        }

        voidprintState() {                  // Ceci est une deuxième méthode
                if (open) System.out.println( " L'interrupteur est ouvert ") ;
                else System.out.println( " L'interrupteur est fermé ") ;
        }
}
```

Nous voyons que :

- Le mot clé **class** permet de définir une classe
- Cette classe contient un seul attribut (une donnée interne). Il est possible de définir autant d'attributs que l'on veut dans une classe.
- Cette classe contient deux méthodes qui permettront d'agir sur les objets appartenant à cette classe. Il est possible de définir autant de méthodes que l'on veut dans une classe.

### 4.1.2 Modificateur de classe : public

Une classe est soit publique, soit privée :

- Une classe publique est visible en dehors du Package dans laquelle elle est définie
- Une classe privée n'est pas visible en dehors de ce Package.

Un Package est un regroupement logique de classes (en fait de fichiers contenant des classes). Nous verrons cette notion en détail dans un chapitre ultérieur…

Par défaut une classe est privée.

Dans un fichier Java, il ne peut y avoir qu'une seule classe publique : Si deux classes publiques sont définies dans un seul fichier il y a une erreur à la compilation.

Une classe est publique si sa définition est précédée du mot clé **public**.

```
public class Interrupteur {                    // Cette classe est publique
        …
}
```

### 4.1.3 Modificateur de classe : final

Une classe définie comme **final** ne peut pas être dérivée (on ne peut pas créer une nouvelle classe héritant de cette classe).

Ce modificateur est utile pour des raisons de sécurité et de performances. De nombreuses classes du JDK sont **final**.

```
public final class Interrupteur {              // Cette classe ne peut être dérivée
        …
}

// La classe suivante dérive de la classe Interrupteur qui est final
class InteruptAvecVoyant extends Interrupteur {      // => Erreur de compilation
…
}
```

## 4.2 Les attributs (== les champs)

Les attributs (ou champs) sont les données contenues dans les objets d'une certaine classe.

### 4.2.1 Déclaration et initialisation

Voici un exemple plus complexe de classe, décrivant des objets représentants des dates et contenant trois attributs :

```
class Date {
        int jour ;                          // Par défaut, un attribut est initialisé à zéro
        int mois = 1 ;
        int an = 2000 ;
        final int MAX_MOIS = 12 ;           // Ceci est une constante : elle doit être initialisée

        Date(int _an) { an = _an; }         // Ceci est un constructeur, méthode spéciale utilisée lors
                                            // de la création d'objets.  Nous verrons cette notion en
                                            // détail plus tard.

        void print() { System.out.println( jour + "/"  + mois + "/" + an ); }
}
        …
        Date d = new Date(1998) ;  // Création d'un nouvel objet de la classe Date, d est
                                            // une référence sur cet objet. Nous reviendrons sur
                                            // ces notions plus tard

                                            // Invocation de la méthode print sur l'objet référencé par d
        d.print() ;                         // => affiche : 0/1/1998
```

Lors de la création d'un objet, ses attributs sont initialisés par défaut :

- Pour les ***valeurs numériques*** : à **zéro**
- Pour les ***références*** (qui désignent des objets) : à **null**
- Pour le ***boolean*** : à **false**

Les attributs peuvent être initialisés de manière spécifique par le programmeur :

- Lors de leur déclaration (comme l'attribut mois)
- Dans le constructeur (comme l'attribut an)

***Remarques*** :

- Un constructeur est une méthode spéciale qui est appelée lors de la création d'un objet.
- Un objet est créé avec l'opérateur **new**, qui renvoie une référence sur cet objet.

Nous reviendrons sur ces notions ultérieurement.

***Attention*** : Seuls les attributs sont initialisés automatiquement : toute variable locale (à une méthode) devra être initialisée avant son utilisation, sinon le code ne pourra pas être compilé.

### 4.2.2 Le mot clé : final

Le mot clé **final**, utilisé pour un attribut, permet de spécifier que cet attribut est une constante (sa valeur ne pourra jamais être modifiée). Un attribut **final** doit obligatoirement être initialisée lors de sa déclaration (puisqu'on ne peut pas le modifier après cela ) !

### 4.2.3 Le mot clé : static

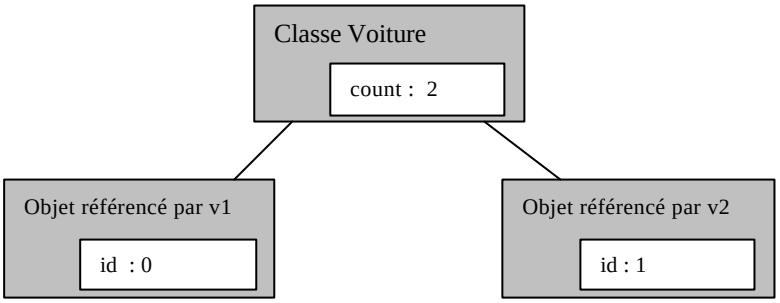
Le mot clé **static**, utilisé pour un attribut, permet d'indiquer que cet attribut est commun à tous les objets de la classe concernée : il s'agit d'un attribut de la classe elle-même, et si on modifie cet attribut pour un objet donné, il sera modifié pour tous les objets de la classe (puisque c'est le même).

```
class Voiture {
        static int count = 0 ;
        int id ;

        Voiture() { id = count++ ; }
}


        …
        Voiture v1 = new Voiture() ;
        Voiture v2 = new Voiture() ;
        …
```

En mémoire :




Lorsque l'on crée un nouvel objet d'une certaine classe, la machine virtuelle se charge de réserver de la mémoire et d'initialiser chaque attribut "normal" de cette classe. Par contre **un attribut de classe** (**static**) est créé et initialisé une fois pour toutes lors de la définition de cette classe.

```
Class MathUtil {
        final static float PI = 3.14 ;
}


        …
        diam = 2 * MathUtil.PI * rayon
        ….
```

Un **attribut de classe** peut être accédé par n'importe quel nom de référence associé à un objet de cette classe, ou par le nom de la classe elle-même.

## 4.3 Les conventions de nommage

Il est très important de disposer de conventions communes et cohérentes de nommage (pour savoir immédiatement ce que représente un nom, pour que les autres puissent comprendre facilement votre code, …).

Les conventions habituellement utilisées dans Java sont :

- Le nom des ***packages***, ***sous-packages***, etc.… est **écrit en minuscule**, et ne comporte généralement qu'un mot significatif (Ex : java.io)
- Le nom des ***classes*** **commence par une majuscule, suivie de minuscules**, **une majuscule apparaissant dans chaque nouveau mot** constituant ce nom (Ex : StringBufferInputStream).

- Le nom des ***méthodes*** et ***attributs*** **débute par une minuscule, suivie de minuscules, une majuscule apparaissant dans chaque nouveau mot** constituant ce nom (Ex : getAppletInfo() ).
- Le nom des **variables locales** est **écrit en minuscule**.
- Le nom des ***attributs constants*** est **écrit en majuscule**.

## 4.4 Les méthodes

Une méthode peut être considérée comme une fonction qui est membre d'une certaine classe : Les méthodes Java sont syntaxiquement très proches des fonctions C : elles ont un nom, peuvent accepter des paramètres et retourner une valeur.

### 4.4.1 Le passage des paramètres

Il est très important de comprendre la différence entre le comportement des types de base et celui des objets lors du passage de paramètres.

#### 4.4.1.1 Les types de bases : passage de paramètres par valeur

Les types de base sont toujours passés par valeur : lors de l'appel d'une méthode, la valeur des paramètres correspondants à des types de base est dupliquée et ce sont ces valeurs dupliquées qui sont manipulées dans la méthode. Les variables utilisées lors de l'appel ne sont donc jamais modifiées par celui-ci.

```
class Point {
        private int x, y ;

        public void setXY(float newx, float newy) { x = newx++ ; y = newy++ ; }
}


        …
        int a = 0 , b = 0 ;
        Point p = new Point() ;
        p.setXY(a,b) ;
        // a et b valent toujours zéro…
```

***Remarque*** : Nous verrons la signification de **public** et **private** dans un prochain chapitre.

#### 4.4.1.2 Les objets : passage de paramètres par adresse

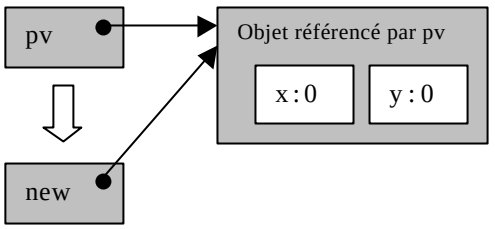
Lorsque l'on passe en paramètre un objet, c'est la référence sur cet objet qui est dupliqué. Il n'y a donc pas duplication de l'objet lui-même. Si la méthode modifie l'objet, il sera donc toujours modifié après l'appel à cette méthode.

```
class PointValue {
        public int x,y ;
}

class point {
        private int x, y ;

        public void setXY(PointValue new) { x = new.x++ ; y = new.y++ ; }
}

        …
        PointValue pv = new PointValue()  ;
        Point p = new Point() ;
        pv.x = 0 ;
        pv.y = 0 ;
        p.setXY(pv) ;
        // pv.x et pv.y valent maintenant un…
```




### 4.4.2 La surcharge des méthodes

La surcharge permet de définir des méthodes portant le même nom, mais acceptant des paramètres de type différent et/ou en nombre différent. Il est possible de considérer que l'on obtient ainsi plusieurs versions de la même méthode.

En fonction du type et du nombre des paramètres passés lors de l'appel, c'est une version ou une autre de la méthode qui sera effectivement appelée.

***Attention*** : Le type de retour d'une méthode ne permet pas de différencier deux méthodes portant le même nom (si c'est la seule différence).

```
class Test {
        void print(int i) { … }
        void print(float f) { … }
        void print(int i, int j) { … }
}
        …
        Test t = new Test() ;
        int i ;
        t.print( i ) ;  // => La première méthode est appelée
```

A l'appel, la machine virtuelle Java détermine quelle est la méthode dont la lise de paramètres est la plus proche des paramètres effectivement envoyés. Dans le cas d'un appel de méthode avec des types ne correspondants pas exactement, des conversions implicites peuvent être effectuées.

```
        short s ;
        t.print( s ) ;  // => La première méthode est appelée avec une conversion de s en int
```

Dans le cas où plusieurs méthodes pourraient correspondre, la machine virtuelle va déterminer pour chacune d'entre elles le coût des conversions nécessaires à partir du tableau suivant. Elle choisira ainsi la méthode correspondant au plus faible coût.

Si le coût global des conversions des paramètres passés dépasse 10, le compilateur refusera l'appel. Il faudra alors expliciter cette conversion.

**Coûts des conversions**

| *de \ vers* | byte | short | char | int | long | float | double |
|---|---|---|---|---|---|---|---|
| **byte** | 0 | 1 | 2 | 3 | 4 | 6 | 7 |
| **short** | 10 | 0 | 10 | 1 | 2 | 4 | 6 |
| **char** | 11 | 10 | 0 | 1 | 2 | 4 | 5 |
| **int** | 12 | 11 | 11 | 0 | 1 | 5 | 4 |
| **long** | 12 | 11 | 11 | 10 | 0 | 6 | 5 |
| **float** | 15 | 14 | 13 | 12 | 11 | 0 | 1 |
| **double** | 16 | 15 | 14 | 13 | 12 | 10 | 0 |

### 4.4.3 Le mot clé : static

Il est possible de définir une méthode de type **static**. De même que pour les attributs, cela signifie que les actions de cette méthode concernent la classe entière (pas un objet particulier). Pour cette raison, **une méthode static, ne peut accéder que les attributs static** de la classe : lorsque l'on utilise une telle méthode vouloir accéder à un attribut appartenant à un objet particulier n'a pas de sens.

On peut invoquer une méthode **static** soit sur la classe elle-même, soit sur un objet de cette classe. Dans tous les cas elle s'appliquera sur la classe entière.

L'utilisation d'une méthode static ne nécessite pas la construction d'un objet appartenant à la classe correspondante.

```
class MathTool {
        final static double PI = 3.14 ;

        static double getPI() { return PI ; }
        static double diametre( double rayon ) { return 2*PI*rayon ; }
        static double power(double x) { return x * x ; }
}

class Test {
        void methode1() {
                double i = MathTool.power(6) ; /* invocation d'une méthode static sur une classe */
        }
        void methode2() {
                MarhTool tool = new MathTool() ;
                double i = tool.power(6) ;          /* idem mais sur un objet. La première méthode est
                                                    plus logique */
        }
```

***Remarque*** : Cet exemple présente une utilisation classique des méthodes **static**, à savoir une classe rassemblant un ensemble d'outils sur un certain sujet.

Une autre utilisation est de mettre sous forme de méthodes static tout ce qui concerne la gestion des objets appartenant à la classe concernée.

```
class Voiture {
        // Gestion des objets de la classe
        static ListeDeVoiture liste ;

        static Voiture getVoiture( int numero ) {
                … retourne l'objet voiture correspondant, à partir de la liste
                }
        static Voiture newVoiture(int numero) {
                Voiture v = new Voiture(numero) ;
                … insertion de v dans la liste
                }
        // Description du comportement de chaque objet Voiture
        Voiture(int numero) { … }
        void print() { … }
        …
}
// exemple d'utilisation :
        …
        Voiture v1 = Voiture.getVoiture(435) ;        // Appel d'une méthode static sur la classe
        v1.print() ;                                  // Appel d'une méthode normale sur l'objet v1
        …
```

Enfin, la première utilisation du mot clé static concerne la méthode main, dont une version peut être définie pour chaque classe et qui sera la première méthode appelée, si et seulement si on lance la Machine Virtuelle Java directement avec cette classe.

```
class Voiture {
        // Méthode utilisée si cette classe est exécutée directement
        public static void main(String args[]) {
                Voiture v1 = Voiture.getVoiture(435)
                v1.print() ;
        }
        // Gestion des objets de la classe
        static ListeDeVoiture liste ;

        static Voiture getVoiture( int numero ) {
                … retourne l'objet voiture correspondant, à partir de la liste
                }
        static Voiture newVoiture(int numero) {
                Voiture v = new Voiture(numero) ;
                … insertion de v dans la liste
                }
        // Description du comportement de chaque objet Voiture
        Voiture(int numero) { … }
        void print() { … }
        …
}
```

### 4.4.4 Les constructeurs

Un constructeur est une méthode particulière qui permet d'initialiser les objets de la classe concernée. Chaque fois qu'un objet est construit (avec l'opérateur new, comme nous le verrons), un constructeur est automatiquement appelé.

Un constructeur a le même nom que la classe, ne retourne jamais de valeur et peut prendre des paramètres.

Lorsqu'une classe ne comporte pas de constructeur, le compilateur génère un constructeur par défaut qui ne prend aucun paramètre, et qui initialise tous les attributs à zéro.

Il est possible de définir plusieurs constructeurs pour une même classe, à condition qu'ils respectent les règles de surcharge (nombre et / ou type des arguments différents)

```
class Date {
        int _jour =1 ;
        int _mois =1 ;
        int _an =1990 ;

        Date( ) { _an = 2000 ; }
        Date( int an ) { _an = an ; }
        Date( int jour, int mois, int an ) { _jour = jour ; _mois = mois ; _an = an ; }
}
```

La classe précédente contient trois constructeurs. Nous verrons plus tard comment déclencher l'un ou l'autre de ces constructeurs lors de la construction d'un objet.

***Attention*** : Une erreur classique consiste à écrire une méthode ayant le nom de la classe mais aussi un type de retour (par exemple void). Cette méthode est alors prise pour une méthode normale, pas pour un constructeur.

### 4.4.5 Le bloc d'initialisation

Les constructeurs permettent d'initialiser les attributs des objets lors de leur création. Il est nécessaire de disposer d'un mécanisme similaire pour initialiser les attributs static : c'est le bloc d'initialisation.

Le bloc d'initialisation est un bloc d'instruction, précédé du mot clé static et placé dans la définition de la classe. Les instructions qui le composent sont exécutées lorsque la classe est chargée dans la machine virtuelle : il s'agit en quelque sorte du constructeur de la classe d'objet.

```
class  Date {
        static String jours[] = new String[7] ;
        static {
                jours[0] = "Lundi" ;
                jours[1] = "Mardi" ;
                jours[2] = "Mercredi" ;
                jours[3] = "Jeudi" ;
                jours[4] = "Vendredi" ;
                jours[5] = "Samedi" ;
                jours[6] = "Dimanche" ;
                System.out.println("La classe Date est initialisée") ;
        }
        …
}
```

### 4.4.6 La méthode de finalisation : finalize()

Il est possible dans toute classe de définir une méthode qui sera appelée automatiquement avant la destruction d'un objet de cette classe par le Garbage Collector. Cette méthode est unique et doit s'appeler **finalize**().

```
Public class Fichier {
        …
        public Fichier(String path) { /* Ouverture du fichier */ }
        …
        protected void finalize() throws Throwable {
                /* Fermeture du fichier */
                super.finalize() ;
        }
}
```

***Remarque*** : les mots clés **protected**, **throws** et **super** seront expliqués dans la suite de ce cours. Il faut tout de même noter que l'instruction ***super.finalize()*** permet d'appeler la méthode de finalisation de la classe mère de la classe Fichier. Vous comprendrez cette notion dans le chapitre sur l'héritage des classes.

## 4.5 Les références

Les références sont les variables qui permettent de désigner et manipuler les objets. La notion de référence est assez proche de celle de pointeur, mais, contrairement au langage C, il n'est pas possible de manipuler directement l'adresse contenue dans ces pointeurs : cela est donc moins souple, mais beaucoup plus simple et sécurisant qu'un vrai pointeur.

### 4.5.1 La création d'objet

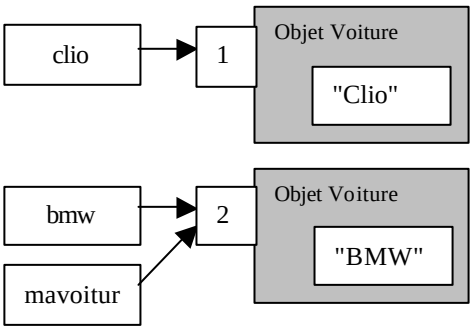
La création d'un objet se fait à l'aide de l'opérateur **new** qui se charge de :

1. Allouer l'espace mémoire nécessaire pour stocker les attributs de l'objet en cours de construction
2. Appeler le constructeur de l'objet adéquat pour initialiser les attributs. Le choix du constructeur se fait en fonction des paramètres passés à l'opérateur new (mêmes règles que pour la surcharge des méthodes normales)
3. Retourner une référence sur l'objet créé.

Les références doivent être considérées comme des pointeurs "intelligents" sur les objets alloués dynamiquement par l'opérateur new : il est possible de copier une référence, et donc de disposer à un instant t de plusieurs références sur un même objet ; Le nombre de référence sur un même objet est mémorisé en permanence dans un compteur ; lorsque celui-ci tombe à zéro (plus personne ne dispose de l'adresse de cet objet) la mémoire est automatiquement libérée par le Garbage Collector

```
1       …
        if ( test) {
2               Voiture clio = new Voiture("Clio") ;
3               Voiture bmw = new Voiture("BMW") ;

                if ( moi.estUnFrimeur()) {
4                       Voiture mavoiture = bmw ;
                        …
5               }
                …
6       }
7
```




| Etapes | Nombre de références |
|---|---|

| | Clio | BMW |
|---|---|---|
| ***1*** | - | - |
| ***2*** | 1 | - |
| ***3*** | 1 | 1 |
| ***4*** | 1 | 2 |
| ***5*** | 1 | 1 |
| ***6*** | 0 | 0 |
| ***7*** | - | - |

Action du Garbage Collector

Le **Garbage Collector** est un outil très puissant pour gérer les problèmes de mémoires en Java. Il est en particulier impossible d'utiliser de la mémoire non libérée et donc d'écraser la mémoire.

### 4.5.2 La référence sur l'objet courant : this

Le mot-clé **this** représente une référence sur l'objet courant (celui qui est en train d'exécuter la méthode dans laquelle se trouvent les instructions concernées).

**this** peut être utile :

- Lorsqu'une variable locale (ou un paramètre) "cache", en portant le même nom, un attribut de la classe.
- Pour déclencher un constructeur depuis un autre constructeur.

```
class Date {
        int jour =1 ;
        int mois =1 ;
        int an =1990 ;

        Date( ) { an = 2000 ; }              /* peut aussi s'écrire  : this.an = 2000 */

        Date( int an ) {
                this.an = an ;               /* Le paramètre an cache l'attribut an */
                 }

        Date( int jour, int mois, int an ) {
                this.jour = jour ;
                this.mois = mois ;
                this(an) ;                   /* appel du deuxième constructeur */
                }
}
```

### 4.5.3 La référence vide : null

Le mot clé **null** permet de représenter la référence qui ne référence rien. On peut assigner cette valeur à n'importe quelle variable ou attribut contenant une référence.

C'est la valeur par défaut d'initialisation des attributs représentant des références et on peut bien entendu tester si une référence est égale à **null**.

```
Class Test {
        Voiture  v1 ;                   /* Initialisée à null par défaut */

        void methode() {
                …
                if ( v1 == null ) v1 = new Voiture("Volvo") ;
                …
                }
}
```

***Attention*** : Si une méthode est invoquée sur une référence égale à null, cela déclenche une erreur du type *NullPointerException*.

### 4.5.4 Les opérateurs sur les références

Les seuls opérateurs sur les références sont des opérateurs logiques :

| | |
|---|---|
| **==** | : permet de tester si deux références désignent le même objet. |
| **!=** | : permet de tester si deux références ne désignent pas le même objet. |
| **instanceof** | : permet de tester si l'objet référencé est une instance d'une classe donnée ou d'une de ses sous-classes |

```
if ( refDeTypeInconnu instanceof Voiture ) {
        … Traitement pour une voiture
}
```

## 4.6 Les tableaux

Un tableau dans Java est toujours un objet, même si les données de ce tableau appartiennent à un type de base.

Un tableau permet de contenir plusieurs éléments du même type (type de base ou classe). La taille d'un tableau est fixée définitivement à sa création (donc après la compilation, contrairement au C).

Un tableau étant un objet, il doit être alloué avec l'opérateur **new**, il est détruit automatiquement par le Garbage Collector lorsqu'il n'est plus référencé et il est systématiquement passé par référence lors de l'appel d'une méthode.

***Remarque*** : Pour passer un type de base par référence, il suffit donc de passer un tableau de un élément basé sur ce type de base.

```
// tableau d'entiers
int tab [] = new int[10] ;

// Tableau de 5 références initialisées à null
Voiture tv[] = new Voiture[5] ;
```

Il existe deux syntaxes pour déclarer un tableau :

La syntaxe de C : **int tab[] ;**
La syntaxe pure Java : **int[] tab ;**

La syntaxe d'initialisation d'un tableau de type de base est la même que celle du langage C.

Il est possible d'utiliser des tableaux multidimensionnels (même syntaxe que le C)

```
int       tab1[]   = new int [3] ;      /* syntaxes équivalentes */
int[]     tab2     = new int [3] ;

int[]     tab3     = {1, 2, 3 } ;       /* initialisation */
int[][]   tab4     = new int[3][8] ;    /* multidimension */
```

Il est possible pour une méthode de préciser un type de retour "tableau" :

**int[] methode(…) { … }**

ou **int methode(…) [] { … }**

La syntaxe pour accéder un élément d'un tableau est la même que pour le langage C. En particulier les indices d'un tableau vont de zéro à la taille du tableau moins un.

***Attention*** : Lorsque l'on déclare et crée avec l'opérateur new un tableau de 10 int (ou de n'importe quel type de base), le tableau contient 10 entiers initialisés à zéro par défaut. Lorsqu'il s'agit d'un tableau de référence sur des objets, celui-ci contient 10 références initialisées à null par défaut, il reste encore à créer 10 objets pour donner un sens à ces références.

```
int[] newTab(int dim) {
        int[] tab = new int[dim] ;
        for (int i ; i<dim ; i++ } tab[i] = i ;
        return tab ;
}
…
int[] t = obj.newTab(15) ;
```

Tout tableau (qui est un objet) possède l'attribue **length** qui contient la dimension du tableau. De ce point de vue un tableau multidimensionnel est considéré comme un tableau de tableaux, de tableaux, …

```
void printTabDim1 (int[] tab) {
        for (int i=0 ; i < tab.length ; i++ ) System.out.println(tab[i]) ;
}

void printTabDim2 (int[][] tab) {
        for (int i=0 ; i < tab.length ; i++ )
                for (int j=0 ; j < tab[i].length ; j++) System.out.println(tab[i][j]) ;
}
```

# 5. L'héritage

Comme nous l'avons vu dans le premier chapitre, il est possible de dériver une classe à partir d'une autre classe : la **classe fille** ainsi créée (ou **sous-classe,** ou **classe dérivée**) hérite alors des caractéristiques de sa **classe mère** (ou **super-classe**), tout en la spécialisant avec de nouvelles fonctionnalités. Le but de ce chapitre est de voir comment ce principe est mis en application dans Java.

## 5.1 Principes généraux, le mot clé extends

**Toute classe dans Java est une sous classe de la classe Object** (elle en hérite directement ou indirectement et éventuellement implicitement).

Une classe ne peut hériter directement que d'une seule classe : il n'y a **pas d'héritage multiple** dans Java contrairement à C++. Cette caractéristique du langage permet de supprimer de nombreux problèmes liés à l'héritage multiple et est compensée par la notion d'interface que nous aborderons dans un prochain chapitre. Il est cependant possible d'avoir autant de niveau d'héritage que voulu : une classe peut avoir une mère, une grand-mère, une arrière-grand-mère, etc.

Une sous-classe peut redéfinir une ou plusieurs des méthodes dont elle hérite. Bien entendu, elle peut aussi en définir de nouvelles, ainsi que de nouveaux attributs.

Il est possible d'**interdire qu'une classe puisse être héritée** : il suffit de déclarer cette **classe final** (déjà vu dans le chapitre précédent)

Il est possible d'**interdire qu'une méthode soit redéfinie** dans une sous-classe : il suffit de déclarer cette **méthode final** (déjà vu).

Le mot clé **extends** permet de décrire les relations d'héritage : pour créer une nouvelle classe fille, il suffit de déclarer cette classe en faisant suivre son nom du mot clé **extends** puis du nom de sa classe mère.

Plutôt que de faire des discours, voici un exemple classique de hiérarchie de classes.

```
class Felin {
        boolean afaim = true ;
        void parler() { }
        void appeler() {
                System.out.println("Le Félin est appelé") ;
                if (afaim) parler() ;
        }
}
final class Chat extends Felin {
        String race ;
        void parler() { System.out.println("miaou!"); }
}
final class Tigre extends Felin {
        void parler() { System.out.println("Groar!"); }
        void chasser() { … }
}

        …
        Tigre tigre = new Tigre() ;
        tigre.chasser() ;           // OK
        tigre.appeler() ;           // OK : méthode héritée de la classe Felin
```

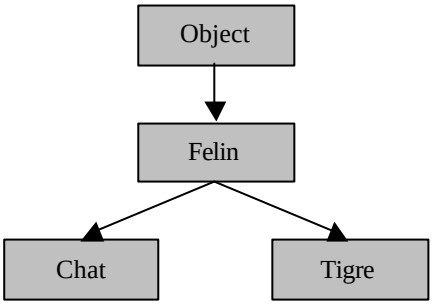
Remarques :

- Les classes *Chat* et *Tigre* héritent de la classe *Felin*, ce qui signifie que les instances de *Chat* et *Tigre* seront aussi des *Félins.* En particulier les deux classes filles héritent de l'attribut *afaim,* et des méthodes *parler()* et *appeler()*.

- On remarque dans chaque classe fille que la méthode *parler()* est redéfinie et que de nouveaux attributs et méthodes sont définis (*race*, *chasser()*). Les attributs d'une classe dérivée comprennent bien les attributs hérités ET des attributs propres (idem pour les méthodes).

- Les classes *Chat* et *Tigre* ont été déclarées **final** : on ne pourra donc pas définir de nouvelle classe héritant d'une de ces classes.

- La class Felin n'a explicitement aucune classe mère. Elle hérite donc implicitement de la classe Object.




```
Tigre  tigre = new Tigre() ;
Felin  felin ;

felin = tigre ;          // OK : c'est une conversion implicite, en fait la classe Tigre héritant de Felin,
                         // tout Tigre est aussi un Felin

tigre = felin ;          // Erreur à la compilation : Tous les félins ne sont pas des tigres
                         //          => une conversion explicite est obligatoire

tigre = (tigre)felin ;   // OK
```

Une variable qui référence un objet d'une certaine classe peut référencer un objet de n'importe laquelle de ses sous-classes (un objet membre d'une sous-classe est aussi membre de la super-classe) : c'est le cas ici pour la variable qui référence un objet de type Felin qui peut donc contenir la référence sur un Tigre

```
Felin felin ;
Tigre tigre = new Tigre() ;
felin = tigre ;
felin.parler() ;          // la référence felin est de type Felin, mais l'objet réellement référencé est de
                          // type Tigre  => c'est la méthode Tigre.parler() qui est réellement appelée
felin.chasser() ;         // Erreur à la compilation : La méthode n'existe pas dans la classe Felin
                          // (bien que définie dans Tigre)
tigre = felin ;           // Erreur à la compilation : Conversion explicite nécessaire

tigre = (tigre)felin ;    // OK
tigre.parler();           // OK
tigre.chasser();          // OK

Chat chat = new Chat() ;
felin = chat ;
lion = (lion)felin ;      // Erreur détectée lors de l'exécution : ClassException
                          // (impossible à détecter par le compilateur)
```

Le fait de déclencher la methode parler() sur la référence de type Felin, déclenche en fait la méthode de la classe Tigre. C'est un des grands avantages des langages objets : la méthode est exécutée par l'objet réel se trouvant au bout de la référence et lorsqu'on demande à un Tigre de parler il le fait comme on le lui a appris, même si celui qui fait la demande ne sait pas exactement qui il est (un Felin).
Cette capacité de Java s'appelle le **polymorphisme**.

## 5.2 Le Polymorphisme

Une méthode polymorphe est une méthode déclarée dans une super-classe et redéfinie par une sous-classe. Dans Java, toute méthode est par défaut polymorphe (au contraire de C++).
Les méthodes **final** ne peuvent être redéfinies et ne sont donc pas polymorphes (définir une méthode final est utile pour optimiser le byte code et pour des raisons de sécurité).

Comme nous l'avons vu à la fin du chapitre précédent, l'intérêt des méthodes polymorphes et que lorsqu'on les appelle, la version qui est exécutée est toujours celle correspondant à l'objet réel se trouvant au bout de la référence utilisée, même si la nature de cet objet réel n'est pas connue au moment de l'appel.

Typiquement dans un langage qui ne serait pas objet, on effectue couramment le traitement suivant : "je teste le type de l'objet, et en fonction de ce type, je lui demande d'effectuer tel ou tel traitement". Avec le polymorphisme, un objet sait lui-même comment il doit effectuer une tache, et je n'ai donc pas à lui demander ni même a connaître l'existence de son type***. Je dis à un objet ce qu'il doit faire, mais pas comment il doit le faire***…

```
class Zoo {
        int MAX = 10 ;
        Felin[] liste ;
        Int compteur = 0 ;
        Cirque( int taille ) { MAX = taille ; liste = new Felin[MAX] }

        void addFelin( Felin newfelin) {
                if (compteur < MAX)  liste[compteur++] = newfelin ;
                else /* traitement d'erreur */
        }
        final void appeler() {          // cette méthode ne peut être redéfinie dans une sous-classe
                for ( int i=0 ; i<compteur ; i++ ) liste[i].parler() ;          // appel polymorphe
        }
}
        …
        Zoo zoo = new Zoo(10) ;
        zoo.addFelin(new Tigre()) ;
        zoo.addFelin(new Chat()) ;
        …
        zoo.appeler() ;
        …
```

Cet exemple montre une mise en œuvre simple du polymorphisme. La méthode *parler()* a été redéfinie dans toutes les sous-classes de la classe *Felin*. Lorsque l'on demande à chacun des félins de l'application d'exécuter la méthode *parler()*, c'est à chaque fois une méthode différente qui est déclenchée.

## 5.3 Accès à la super-classe d'une classe : super(…)

Le mot clé **super** permet d'accéder à la version non redéfinie des attributs et méthodes de la super-classe. Il permet aussi, s'il est utilisé comme première instruction dans un constructeur, d'appeler un des constructeurs de la classe mère, avec ou sans paramètres.

```
class Mere {
        int attribut ;
        Mere() { attribut = 1 ; }
        Mere(int attribut) ( this.attribut = attribut ; }
        void print() { System.out.println("base" + attribut) ; }
}
class fille extends Mere {
        boolean flag ;
        Fille( int a ) {    super(a) ;
                            flag = true ;
                            }

        void print() {      super.print() ;
                            System.out.println("dérivée") ;
                            }
}
        …
        Fille f = new Fille(2) ;
        Mere m = f ;
        m.print() ;
// Affiche :
dérivée
base 2
```

## 5.4 Méthodes et classes abstraites : abstract

Une méthode abstraite est une méthode dont on donne le prototype, sans en décrire l'implémentation (sans définir les instructions qui décrivent ses actions). Elle doit de plus être précédée du mot clé **abstract**. Une méthode abstraite doit obligatoirement être redéfinie dans une sous-classe pour pouvoir être utilisée (par polymorphisme).

Une classe abstraite est une classe dont la définition est précédée du mot clé **abstract**. Une classe abstraite ne peut jamais être directement instanciée avec l'opérateur **new** pour créer un nouvel objet : seules des classes filles non abstraites peuvent l'être.
Lorsqu'une classe possède une ou plusieurs méthodes abstraites, elle doit elle-même être déclarée abstraite, mais la réciproque n'est pas vraie : une classe peut être abstraite sans pour autant posséder de méthode abstraite.

Les classes et les méthodes abstraites permettent de définir des fonctionnalités, sans spécifier la façon dont ces fonctionnalités sont implémentées.
Une classe héritant d'une classe abstraite doit donner une implémentation à toutes les méthodes abstraites de sa super-classe. Si ce n'est pas le cas, elle doit aussi être déclarée abstraite.

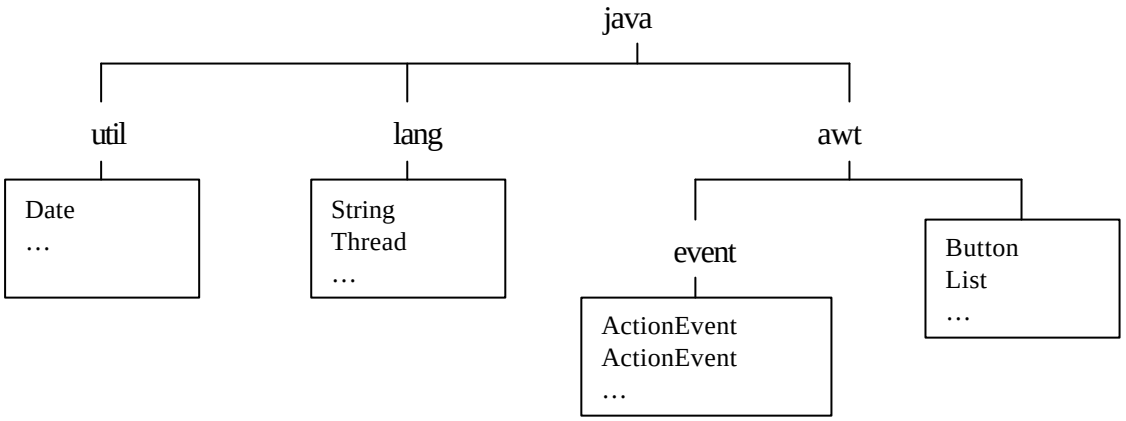
```
abstract class Felin {
        boolean afaim = true ;
        abstract void parler( ) ;
        void appeler() {
                System.out.println("Le Félin est appelé") ;
                if (afaim) parler() ;
        }
}
class Tigre extends Felin {
        void parler() { System.out.println("Groar!"); }
}
```

# 6. Les packages et l'encapsulation

## 6.1 Les packages

Un package peut être considéré comme une bibliothèque de classes : Il permet de regrouper un certain nombre de classes qui sont proches dans une seule famille et ainsi d'apporter un niveau de hiérarchisation supplémentaire au langage.

Les packages sont eux-mêmes organisés hiérarchiquement. : on peut définir des sous-packages, des sous-sous-packages, …




Dans cet exemple :

- La classe **Button** appartient au package **java.awt** (sous-package du package **java**)
- La classe **ActionEvent** appartient au package **java.awt.event**
- Le package **java** ne contient que des sous-packages : **lang** et **awt**

Il y a deux manières d'utiliser une classe stockée dans un package :

- En utilisant le nom du package suivi du nom de la classe

```
java.util.Date now = new java.util.Date() ;
System.out.println(now) ;
```

- En utilisant le mot clé **import** pour importer (inclure) le package auquel appartient la classe

```
import java.util.Date ;          // Doit être en tête du fichier !!
                                 // Ne permet d'importer que la classe Date de java.util
…
Date now = new Date() ;
System.out.println(now) ;
```

```
import java.util.* ;             // Permet d'importer toutes les classes de java.util
…
Date now = new Date() ;
System.out.println(now) ;
```

On peut généralement utiliser l'une ou l'autre de ces deux méthodes, mais il existe un cas ou l'on doit obligatoirement utiliser la première : si deux classes portant le même nom sont définies dans deux packages différents.

**Java importe automatiquement le package java.lang** qui permet d'utiliser des classes comme **Thread** ou **System**.

***Remarque*** : **System** est une classe de **java.lang**, **out** est une variable statique de cette classe (de type **PrintStream**) et **println** est une méthode de la classe **PrintStream**.

## 6.2 La structure de stockage des classes et des packages

Les fichiers sources Java ont pour extension **.java**, les fichiers compilés (donc contenant du byte code) ont pour extension **.class**.

Seules les classes **public** sont accessibles d'un autre package, ou d'un autre fichier (les autres classes ne sont pas connues en dehors de leur fichier). Il ne peut y avoir qu'une et une seule classe **public** dans un fichier **et cette classe doit porter le même nom que le fichier** (en respectant les minuscules / majuscules).

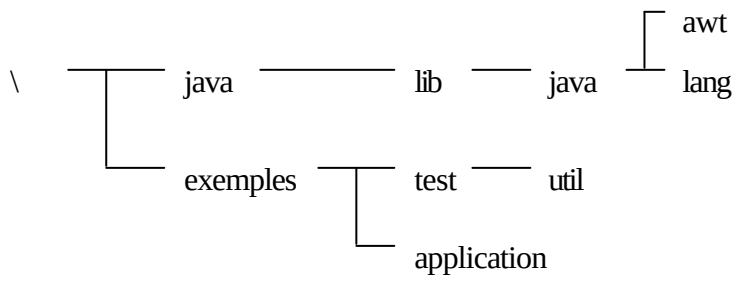
Les fichiers des classes qui font partie d'un package doivent être placés dans une hiérarchie de répertoires correspondant à la hiérarchie des packages. Ils doivent obligatoirement commencer par une déclaration précisant le nom de leur package, précédé du mot clé **package**.

```
// Fichier Classe1.java dans  le répertoire test/util/
package test.util ;
public class Classe1 {
        public void test() { … }
}
```

```
// fichier Appli1.java dans le sous répertoire application/
// la déclaration de package est inutile, puisque cette classe n'est pas destinée à être utilisée par d'autres
import test.util.* ;
public class Appli1 {
        public static void main(String args[]) {
                Classe1 c = new Classe1() ;
                c.test() ;
        }
}
```

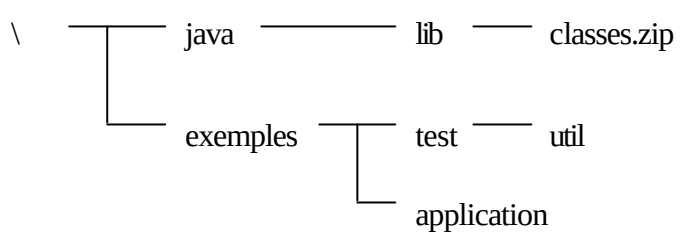
Le compilateur et la machine virtuelle Java utilisent la variable d'environnement **CLASSPATH** pour localiser les packages et donc les classes présentes sur le disque. Cette variable permet de référencer tous les répertoires servant de racine à une arborescence de packages.

Supposons que l'on ait la structure de disque suivante :




Nécessite la définition : **Set CLASSPATH=\java\lib;\exemples**

On a le droit de placer une arborescence de classes dans un fichier archive (fichier zip, compressé ou non, ou archive jar, propre à java). La variable CLASSPATH doit alors référencer ce fichier, avec son chemin complet :




Nécessite la définition : **Set CLASSPATH=\java\lib\classes.zip;\exemples**

## 6.3 Les principaux packages du JDK

| | |
|---|---|
| **java.applet** | Classes de base pour les Applets |
| **java.awt** | Interface Utilisateur |
| **java.io** | Entrées/Sorties, fichiers, … |
| **java.lang** | Classes faisant partie du langage |
| **java.math** | Utilitaires mathématiques |
| **java.net** | Accès au réseau |
| **java.security** | gestion de la sécurité |
| **java.sql** | accès aux bases de données |
| **java.util** | Conteneurs, dates, … |
| **java.util.zip** | Pour compresser/décompresser |

## 6.4 Les règles de visibilité des attributs et méthodes : public, protected, private, friendly

Nous avons déjà abordé les règles de visibilité des classes (public : accès à tous, private : accès uniquement pour les classes du même fichier).

En ce qui concerne les méthodes et les attributs, il existe 4 niveaux de visibilité qui permettent de définir si telle méthode ou tel attribut est visible pour les classes du même package, pour celles en dehors du package courant et pour les classes dérivées.

***La visibilité* friendly *est celle prise par défaut.***

| Accessible aux : | **méthodes de la même classe** | **classes dérivées dans le même package** | **classes du même package** | **classes dérivées dans un autre package** | **classes des autres packages** |
|---|---|---|---|---|---|
| **public** | X | X | X | X | X |
| **protected** | X | X | X | X | |
| **friendly** | X | X | X | | |
| **private** | X | | | | |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | |

***Remarque*** : Les règles de visibilité fonctionnent au niveau des classes, pas des objets. Un objet peut parfaitement accéder à la méthode privée d'un autre objet si celui-ci est de la même classe !

Pour respecter les concepts de l'approche objet, il est nécessaire de respecter les règles de conception suivantes:

1. ***Les attributs doivent toujours être privés***
2. ***Les attributs constants (final) peuvent être publics ou privés***
3. ***Les méthodes à usage strictement interne doivent être privées***
4. ***Si un attribut doit pouvoir être accessible de l'extérieur : définir des méthodes publiques permettant d'y accéder.***

Ces règles permettent de garantir le respect de l'encapsulation des objets: pour utiliser un objet, peut importe de quoi il est fait, seul importe ce qu'il peut faire (les méthodes publiques)…

```
class Personne {
        private String nom ;
        …
        public Personne(String nom) { this.nom = nom ; }
        public String getNom() { return nom ; }
}


        …
        Personne moi = new Personne("Toto") ;

        System.out.println( moi.getNom() ) ;
```

## 6.5 Les classes imbriquées et les classes anonymes

Il est possible de définir des classes dans des classes. Il existe deux types de classes ainsi définies :

- **Les classes imbriquées** : Elles possèdent un nom et sont définies au même niveau qu'une méthode ou un attribut.
- **Les dasses anonymes** : Elles ne possèdent pas de nom et sont définies là où elles sont utilisées, c'est à dire dans le code.

Dans tous les cas la visibilité de ces classes est limitée aux méthodes de la classe dans laquelle elles sont définies. Par contre elles ont accès à tous les éléments de cette classe (même privés).

Exemple de classe imbriquée :

```
public class Voiture {
        class Roue {
                private String modele ;
                Roue(String modele) { this.modele = modele ; }
        }
        private Roue[] roues = new Roue[4] ;
        private int puissance =10 ;

        public Voiture(String modele_roue, int puissance) {
                this.puissance = puissance ;
                for ( int i=0 ; i<roues.length ; i++ ) roues[i] = new Roue(modele_roue) ;
        }
}
```

Exemple de classe anonyme :

```
class Commande {
        public void go() { System.out.println("Pas de commande") ; }
}

class Shell {
        private Vector commandes = new Vector() ; // Objet permettant de contenir N objets

        public void addCommande( Commande commande) {
                commandes.addElement(commande) ;
        }
        public void executer() {
                for ( /* Parcours de tous les éléments */ )
                        ((Commande)e.nextElement()).go() ;
        }
}


class Principale {
        Shell shell = new Shell() ;
        public static void main(String[] args) {
                shell.addCommande( new Commande() {
                                        public void go() { System.out.println("commande") ;}
                                        } ) ;
                …
                shell.executer() ;
        }
}
```

# 7. Les interfaces

## 7.1 Définitions, les mots clés interface et implements.

Une interface est une déclaration permettant de décrire un ensemble de méthodes abstraites et de constantes. On peut considérer une interface comme étant une classe abstraite ne contenant que des méthodes abstraites et que des attributs final et static.

Il est donc impossible d'instancier directement une interface. Par contre, une classe peut implémenter une ou plusieurs interfaces, en utilisant le mot clé **implements** suivi du nom de chaque interface concernée. Cette classe doit alors fournir une version pour chaque méthode déclarée dans les interfaces implémentées.

On peut considérer l'implémentation des interfaces comme une forme d'héritage multiple léger.

Une interface est définie de manière similaire à une classe, mais est caractérisée par le mot clé **interface**.

```
interface Printable {
        void print() ;
}

class Point extends Object implements Printable {
        private double x,y ;
        …
        void print() {
                System.out.println( "(" + x + "," + ")" ) ;
        }
}
```

La notion d'interface permet de mieux découpler l'implémentation d'un objet de la vision que les autres objets peuvent en avoir : une classe peut implémenter plusieurs interfaces, c'est à dire rendre plusieurs types de service (ceux définis au niveau de chaque interface). Réciproquement, la même interface peut être implémentée par plusieurs classes : chaque objet de ces classes fournit donc le même type de service (celui défini dans l'interface).

Nous verrons qu'il est possible de manipuler des références correspondant à une certaine interface, et donc de manipuler des objets à travers ces références en ayant uniquement la connaissance qu'ils implémentent cette interface.

Les attributs déclarés dans une interface sont obligatoirement des constantes statiques : les mots clés static et final ne sont pas nécessaires pour le préciser. Ces attributs doivent donc obligatoirement être initialisés.

```
interface Interval {
        int MIN = 0 ;
        int MAX = 1000 ;
}


        …
        for (int i = Interval.MIN ; i < Interval.MAX ; i++ ) { … }
        …
```

## 7.2 Interfaces et héritage

Nous avons déjà dit qu'une classe peut implémenter plusieurs interface.

Une interface peut aussi hériter d'une ou plusieurs interfaces : elle hérite alors de l'ensemble des méthodes abstraites et constantes de ses ancêtres.

```
interface Printable {          /* exprime le fait de pouvoir être imprimé */
        void print() ;
}

interface InputStream { /* exprime le fait de pouvoir être une source de caractères */
        public int read() ;
}
interface OutputStream { /* exprime le fait de pouvoir accepter des caractères */
        public void write(int) ;
}

interface DataStream extends InputStream, OutputStream {
        public double readDouble() ;
        public void writeDouble(double) ;
}


class MonStream implements DataStream, Printable {
        void print() { …}
        public int read() { …}
        public void write(int) { …}
        public double readDouble(){ …}
        public void writeDouble(double) { …}
}
```

## 7.3 Les références de type interface, l'opérateur instanceof.

Une interface définie un nouveau type : il est possible de définir des références du type d'une interface, c'est à dire permettant de référencer des objets dont tout ce que l'on sait est qu'ils implémentent cette interface.

Des objets complètement différents peuvent donc être référencés et utilisés de la même manière à condition qu'ils implémentent la même interface : c'est une généralisation du polymorphisme et de ses avantages.

L'exemple suivant met en œuvre le polymorphisme au travers de l'utilisation d'une interface. Cette interface définie un service (Persistent) qui est implémenté par un certain nombre de classes de l'application (la classe Client dans l'exemple).
On stocke une collection d'objet implémentant le service Persistent dans un tableau de type Persistent. Chacun de ces objets dispose donc de la méthode save(), avec sa propre implémentation. La classe PersistentManager peut alors manipuler tous ces objets avec l'interface Persistent, sans rien savoir de leur nature réelle.

```
interface Persistent { void save() ; }

class Client extends Personne implements Persistent {
        private int numero ;
        …
        public void save() {
                /* Enregistrement sur fichier ou Base de donnée */
        }
}

class PersistentManager {
        private final static int MAX=100 ;
        private static Persistent[] liste_des_objets = new Persistent[MAX] ;
        static  int nb = 0 ;
        …
        public static void addPersistent(Persistent objet) {
                if (nb<MAX) liste_des_objets[nb++] = objet ;
        }

        public static void saveAll () {
                for (int i=0 ; i<liste_des_objets.length ; i++)
                        if ( liste_des_objets[i] != null ) liste_des_objets[i].save() ;
        }
}


        …
        Client client = new Client("Toto",4529) ;
        PersistentManager.addPersistent(client) ;
        …
        PersistentManager.saveAll() ;
        …
```

L'opérateur **instanceof** peut être utilisé pour savoir si un objet implémente une interface donnée :

```
Point point = new Point() ;
…
if ( point instanceof Printable ) ((Printable)point).print() ;
…
```

# 8. Les exceptions

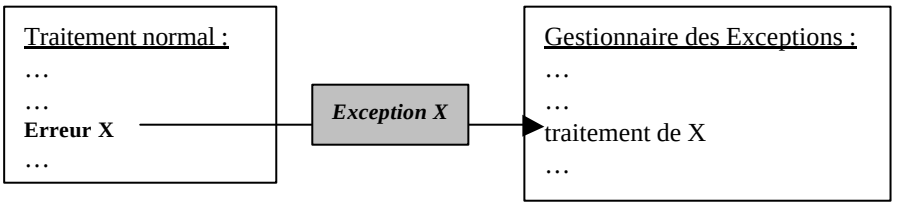
## 8.1 Pourquoi des exceptions ?

Lorsque l'on examine le code source d'un programme écrit en langage C, on peut observer qu'une grosse partie de ce code est constitué de lignes d'instructions concernant la détection et la gestion d'erreurs ou de traitement de cas exceptionnels (division par zéro, impossible d'ouvrir un fichier, mauvaise saisie de données,…). Ces portions de code sont toujours fortement imbriquées dans le reste du code et le programmeur a souvent des difficultés à déterminer ce qu'il doit faire dans tel ou tel cas.

Cela a plusieurs conséquences négatives :

1. Le code est peu lisible, on ne distingue pas le traitement normal des traitements des cas exceptionnels (qui ont souvent une logique très différente)
2. Des traitements d'erreurs sont souvent oubliés par le programmeur.
3. Il est souvent difficile de traiter de manière cohérente l'ensemble des erreurs : comment distinguer un résultat valide d'un cas d'erreur ? (Par exemple si le code retour d'une fonction est utilisé pour transmettre le résultat, où doit-on passer le code d'erreur ?)

C'est pourquoi des langages comme C++ ou Java ont introduit la notion d'exception : un mécanisme facilitant le traitement de tous les cas exceptionnels et permettant en particulier de séparer ces traitements des traitements habituels du programme.
Un cas exceptionnel est représenté par un objet (une exception), qui contient la description du cas exceptionnel, et qui peut être transmis de l'endroit où il a été déclenché jusqu'à celui où l'on sait le traiter (un gestionnaire d'exception).



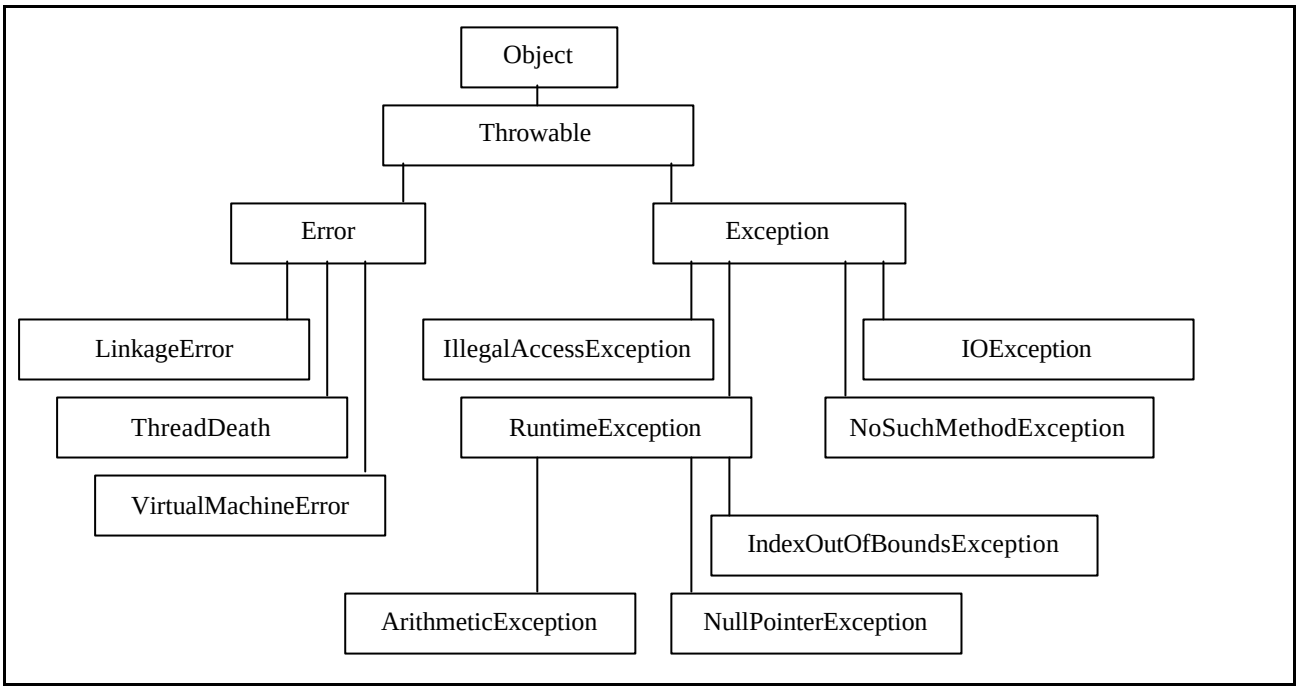

Dans Java, contrairement à C++, le traitement des exceptions est vérifié par le compilateur.

## 8.2 Représentation et déclenchement des exceptions : classe Exception, opérateur throw

Une exception dans Java est un objet appartenant à une classe dérivée de la classe **java.lang.Exception**.

Il existe des exceptions prédéfinies qui dérivent de la classe **java.lang.Error** (pas de **Exception**). Ces exceptions concernent des erreurs internes de la Machine Virtuelle Java : il est fortement déconseillé de les intercepter pour ne pas perturber le fonctionnement de cette machine virtuelle.

Voici une portion de la hiérarchie des exceptions prédéfinies dans Java :




Une exception est déclenchée avec le mot clé **throw**, après avoir créé le nouvel objet **Exception** correspondant.

```
class MonException extends Exception { }
        …
        if ( erreurDétectée ) throw new MonException();
        …
```

## 8.3 Propagation, Capture et traitement des exceptions

Lorsqu'une exception est déclenchée dans une méthode (soit directement avec **throw**, soit parce que une méthode appelée dans cette méthode la déclenche et la propage) il y a deux possibilités :

1. On ne traite pas l'exception localement et on la propage.
2. On traite l'exception localement.

### 8.3.1 Propagation des exceptions : le mot clé throws

Toute exception pouvant être propagée par une méthode doit être signalée dans la déclaration de celle-ci. Pour cela on utilise le mot clé **throws** suivi de la liste des exceptions.

```
class PasDeSolution extends Exception {}

class Equation { /* Equation du second degré ax²+bx+c */
        private double a, b, c ;
        public Equation(double a, double b, double c) { this.a = a ; this.b = b ; this.c = c ; }

        public double resultat() throws PasDeSolution { // Cette méthode propage une exception
                double discriminent  = b*b-4*a*c ;
                if (discriminent < 0) throw new PasDeSolution() ;
                return ( b + Math.sqrt(discriminent) ) / (2 * a) ;
        }
}
```

```
…
void calcul() throws PasDeSolution {
        Equation eq = new Equation(1,0,1) ;
        Eq.resultat() ;
}
// Cette méthode doit déclarer la propagation de l'exception PasDeSolution que Eq.Solution() peut
// déclencher, car elle ne la traite pas localement.
```

### 8.3.2 Capture des exceptions : catch, try et finally

Un gestionnaire d'exception est défini en :

1. Définissant quelles instructions sont surveillées par ce gestionnaire, en plaçant celles-ci dans un bloc d'instructions préfixé par le mot clé **try**.
2. Définissant un ou plusieurs blocs de traitement d'exception préfixés par le mot-clé **catch**.

```
void calcul() {
        try {
                Equation eq = new Equation(1,0,1) ;
                double resultat = Eq.solution() ;
                System.out.println("Resultat = " + resultat) ;
        }
        catch ( PasDeSolution e ) {
                System.out.println("Pas de solutions") ;
        }
}
```

Si l'appel à la méthode *Eq.solution()* se passe mal, l'exécution du bloc **try** est immédiatement arrêtée et le bloc **catch** correspondant est exécuté, puis on passe à la suite du programme. Dans le cas contraire, les instructions du bloc **catch** ne seront jamais prises en compte.

Nous avons vu que les exceptions sont des objets qui font partie d'une hiérarchie de classe basée sur la classe **Exception**. Il est possible de capturer dans un seul bloc **catch** plusieurs exceptions, en capturant une exception dont la classe est une classe mère de toutes les exceptions concernée :

```
        try {
                …
        }
        catch ( Exception e ) { /* permet de capturer tout type d'exception */
                …
        }
```

Il est aussi possible de mettre plusieurs bloc **catch** :

```
try {
        …
}
catch ( IOException e ) { /* permet de capturer tout type d'exception d'entrée/sorties */
        …
}
catch ( PasDeSolution e ) {
        …
}
```

Il est permit dans un bloc **catch** de re-déclencher l'exception interceptée par ce bloc vers un niveau supérieur (si par exemple on s'est aperçu qu'il n'est pas possible de la traiter correctement). Il est alors obligatoire de déclarer cette propagation dans la méthode concernée (sauf si un deuxième bloc try permet de l'intercepter à nouveau).

```
void methode() throw IOException {
        try {
                …
        }
        catch ( IOException e ) { /* permet de capturer tout type d'exception d'entrée/sorties */
                …
                throw e ;
        }
}
```

Il est possible de définir un bloc **finally** qui, contrairement au **catch**, n'est pas obligatoire, et qui permet de spécifier du code qui sera exécuté dans tous les cas, qu'une exception survienne ou pas.
*Ce code sera exécuté même si une instruction **return** est exécutée dans le catch ou le try !*

```
try {
        // Ouvrir un fichier
        // Lire et écrire des données
}
catch ( IOException e ) { /* permet de capturer tout type d'exception d'entrée/sorties */
        System.out.println(e) ;
        return ;
}
finally {
        // Fermeture du fichier
}
…// autres traitements sur les données lues…
```

***Remarque*** : Nous constatons dans cet exemple qu'un objet Exception peut être directement affiché avec la méthode System.out.println(). L'affichage consiste en une chaîne de caractères expliquant la nature de cette exception.

Il est aussi possible en appliquant la méthode **printStackTrace**() sur l'objet exception, d'afficher la pile des appels de méthodes remontées depuis le déclenchement de cette exception :

```
catch ( IOException e ) { /* permet de capturer tout type d'exception d'entrée/sorties */
        System.out.println("Erreur du type : " + e) ;
        e.printStackTrace() ;
}
```

# 9. Les classes de Base

Dans ce chapitre et dans les suivants nous passerons en revue quelques classes indispensables à connaître, sans trop entrer dans les détails : comme pour les bibliothèques de fonctions C, il est impossible et inutile de tout vouloir apprendre sur les classes Java, il y en a trop et il sera toujours indispensable de savoir utiliser le système d'aide et sa propre expérience pour trouver la réponse à nos questions.

Pour chaque classe nous n'aborderons donc que quelques exemples d'utilisation montrant les méthodes les plus importantes à connaître.

## 9.1 La classe Object

La classe Object est la classe mère de toutes les classes, elle contient 2 méthodes que la plupart des classes Java doivent redéfinir :

**String toString()** : qui retourne une description de l'objet.
**boolean equals(Object)** : qui permet de tester l'égalité sémantique de deux objets.

On doit aussi noter l'existence de la méthode **Class getClass()** qui peut être utilisée sur tout objet pour déterminer sa nature (nom de sa classe, nom de ses ancêtres, structure).

```
class Personne  { // dérive de Object par défaut
        private String nom ;
        public Personne(String nom) { this.nom = nom ; }
        public String toString() {
                return "Classe : " + getClass().getName() + " Objet : " + nom ;
        }
        boolean equals(Personne p) { return p.nom.equals(nom) ; }
}

Personne p1 = new Personne("Jean Dupond") ;
Personne p2 = new Personne("Jean Dupond") ;

if ( p1 == p2 ) … // Faux, les références sont différentes
if (p1.equals(p2)) … // Vrai, comparaison sémantique

System.out.println( p1 ) ;
// Affiche : Classe : Personne Objet : Jean Dupond
```

***Remarque*** : Lorsqu'on affiche un objet avec la méthode **println(obj)**, c'est la méthode **toString()** de cet objet qui est utilisée pour afficher son contenu.

## 9.2 Les classes Wrapper

Nous avons vu que les types de base de Java ne sont pas des objets (int, double, …). Les classes Wrapper sont des classes qui permettent de représenter les types de base sous forme d'objets :

| | |
|---|---|
| **Boolean** | pour **boolean** |
| **Integer** | pour **int** |
| **Float** | pour **float** |
| **Double** | pour **double** |
| **Long** | pour **long** |
| **Character** | pour **char** |

Elles permettent en particulier de :

- Récupérer les valeurs minimum et maximum du type de base correspondant.
- Un objet Wrapper peut être créé à partir d'une variable du type de base correspondant ou du contenu d'une chaîne de caractères.
- Les conversions entre types de base et chaînes de caractères sont possibles via des objets Wrapper.
- Les objets Wrapper peuvent être contenus dans des conteneurs (au contraire des types de base).

***Attention*** : les objets Wrapper ne supportent pas les opérateurs classiques (+, *, …)

```
// Exemple d'accès aux valeurs min/max d'un type
        double f ;
        int i ;
        …
        if ( f > Integer.MIN_VALUE and f < Integer.MAX_VALUE) i = (int) f ;
        …
```

```
// Exemple de conversion chaîne => entier (Variante 1)
int stoi(String s ) {
        try { return Integer.parseInt(s) ; }
        catch (Exception e) { return 0 ; }
}
```

```
// Exemple de conversion chaîne => entier (Variante 2)
int stoi(String s ) {
        return (new Integer(s)).intValue() ;
}
```

```
// Exemple de conversion chaîne => entier (Variante 3)
int stoi(String s ) {
        return Integer.valueOf(s).intValue() ;
}
```

```
// Exemple de conversion entier => chaîne (Variante 1)
String itos(int i ) {
        return (new Integer(i)).toString() ;
}
```

```
// Exemple de conversion entier => chaîne (Variante 2)
String itos(int i ) {
        return "" + i ;
}
```

```
// Exemple de conversion entier => chaîne (Variante 3)
String itos(int i ) {
        return String.valueOf(i)  ;
}
```

## 9.3 Les chaînes de caractères

Elles peuvent être représentées par deux classes d'objets : **String** et **StringBuffer**

### 9.3.1 La classe java.lang.String

Cette classe décrit des objets qui contiennent une chaîne de caractères ***constante*** (cette chaîne ne peut pas être modifiée).

C'est une classe particulière qui est reconnue par le compilateur : une constante de type chaîne de caractères est reconnue par celui-ci comme étant un objet de la classe **String**.

```
String Message = "Hello World" ;
```

C'est aussi la seule classe qui dispose d'opérateurs supplémentaires : + et += pour la concaténation de chaînes de caractères.

```
int nombre = 3 ;
String message = "Mon nombre est " + 3 ;    // Ici le nombre est converti automatiquement en String

System.out.println("Mon message est : " + message ) ;
// Affiche : Mon message est : message 3
```

La classe String possède de nombreuses méthodes, voici quelques exemples :

```
Boolean chercheMot( String texte, String atrouver, boolean ignoremajuscules) {
        int ici = 0 ;

        while ( ici < texte.length() ) {
                int suivant = texte.indexOf( ',', ici ) ;
                if ( suivant == -1 ) suivant = texte.length() ;
                String mot = texte.substring( ici, suivant) ;
                if ( ignoremajuscules ) { if ( mot.equalsIgnoreCase( atrouver ) ) return true ; }
                else if ( mot.equals( atrouver ) ) return true ;
                ici = next+1 ;
        }
        return false ;
}
```

Description rapide des méthodes utilisées :

| | |
|---|---|
| **int length()** | Retourne le nombre de caractères compris dans la chaîne. |
| **int indexOf(char c, int i)** | Retourne la position du caractère c en partant de la position i |
| **String substring(int i, int j)** | Retourne une chaîne extraite de la chaîne sur laquelle est appliquée cette méthode, en partant de la position i à la position j |
| **boolean equals(String s)** | comparaison sémantique des chaînes. |

### 9.3.2 La classe java.lang.StringBuffer

La classe **StringBuffer** permet de représenter des chaînes de caractères de taille variable.

Contrairement à la classe **String**, il n'est pas possible d'utiliser l'opérateur + avec les objets de cette classe.

Lorsqu'un objet du type **StringBuffer** est construit, un tableau de caractères est alloué dans l'objet. Sa taille initiale est de 16, mais si l'on ajoute plus de 16 caractères dans l'objet, un nouveau tableau, plus grand est alloué (les caractères sont recopiés de l'ancien tableau vers le nouveau et l'ancien tableau est détruit).

Un objet de la classe StringBuffer se caractérise donc par deux tailles, qui sont retournées par les méthodes :

**int length()** : qui retourne le nombre de caractères réellement contenus
**int capacity()** : qui retourne la taille actuelle du tableau interne

Quelques méthodes supplémentaires:

**StringBuffer append(p)** : ajoute p en fin de chaîne (p est n'importe quel type de base)
**StringBuffer insert(int offset, p)** : idem, mais en insérant p à l'emplacement indiqué par offset
**StringBuffer reverse()** : inversion des caractères de la chaîne.

### 9.3.3 La classe java.util.StringTokenizer

Cette classe permet de construire des objets qui savent découper des chaînes de caractères en sous-chaînes (C'est un objet ayant un caractère utilitaire => package java.util).

Lors de la construction du **StringTokenizer**, il faut préciser la chaîne à découper et, dans une seconde chaîne, le ou les caractères qui doivent être utilisés pour découper cette chaîne :

```
void AfficheParMots(String texte) {
        StringTokenizer st = new StringTokenizer(texte, ",:") ;

        while ( st.hasMoreTokens() ) {
                String mot = st.nextToken() ;
                System.out.println(mot) ;
        }
}
…
AfficheParMots("Lundi,Mardi:Mercredi;Jeudi");
// Affiche :
Lundi
Mardi
Mercredi;Jeudi
```

## 9.4 Les conteneurs

Les conteneurs sont des objets qui permettent de contenir d'autres objets.

Le premier conteneur que nous avons vu est le tableau, qui est un élément de base du langage Java lui-même, et qui peut contenir des références à des objets ou d es types de base.

**Les conteneurs** que nous allons voir dans ce chapitre **permettent de contenir des références à des objets dérivant de la classe Object**, c'est à dire tout objet appartenant à une classe ( *mais* ***pas les types de base, sauf si l'on utilise un Wrapper***).

Les méthodes proposées par les conteneurs (pour accéder à leur contenu) retournent des références de type **Object** que l'on devra souvent **convertir explicitement** vers le type approprié.

### 9.4.1 La classe java.util.Vector

Les objets de la classe **Vector** sont aux tableaux ce que les **StringBuffer** sont aux **String** : Ils représentent des tableaux à taille variable. Le fonctionnement interne de ces deux classes est d'ailleurs très similaire (redimmensionnement automatique du tableau interne si cela est nécessaire).

Les objets de la classe Vector étant à taille variable, il n'y a pas de limite au nombre d'objets qu'il peut contenir (la seule limite est la taille de la mémoire).

Exemple d'utilisation :

```
Vector vec = new Vector() ;

for (int i=0 ; i<10 ; i++) {
        Integer element = new Integer(i) ;
        vec.addElement(element) ;          // Ajout en fin de Vecteur
}
// => 0 1 2 3 4 5 6 7 8 9
…
Integer i = new Integer(15) ;
vec.insertElementAt(i,5) ;          // Insertion à la position indiquée
// => 0 1 2 3 4 15 5 6 7 8 9
…
vec.removeElementAt(0) ;          // Suppression de l'élément indiqué
// => 1 2 3 4 15 5 6 7 8 9
…
Integer j = (Integer)vec.elementAt(6) ;
// j contient une référence sur l'objet Integer contenant 5
…
vec.removeAllElements() ;                    // Suppression de tous les éléments
// =>
```

Autres méthodes disponibles :

| | |
|---|---|
| **Object elementAt(int index)** | : retourne (sans retirer) l'élément à la position index |
| **void setElementAt(Object obj)** | : place l'objet à la position indiquée (remplacement) |
| **boolean contains(Object obj)** | : retourne true si obj est dans le tableau |
| **int indexOf(Object obj)** | : retourne la position de obj (-1 si obj n'est pas présent) |
| **int size()** | : retourne la taille du tableau |

### 9.4.2 L'interface java.util.Enumeration

Un objet implémentant l'interface **Enumeration** permet de parcourir les éléments d'un conteneur (c'est un itérateur de conteneur).

Un tel objet ne contient pas de données mis à part une référence à un conteneur et une "position courante" dans ce conteneur.

**Enumeration** est l'interface unique pour le parcours de tous les types de conteneurs définis dans les classes de base de Java. Un objet implémentant cette interface permet de réaliser une et une seule itération sur le conteneur qui lui est associé.

Pour les conteneurs de type **Vector**, la méthode **elements()** permet de récupérer un objet implémentant l'interface Enumeration. Il est alors possible d'utiliser deux méthodes sur cet objet :

| | |
|---|---|
| **boolean hasMoreElements()** | : Teste s'il reste des éléments à parcourir |

**Object nextElement()** : Retourne l'élément courant et passe au suivant

```
// Soit le conteneur :
Vector vec = new Vector() ;

for (int i=0 ; i<10 ; i++) {
        Integer element = new Integer(i) ;
        vec.addElement(element) ;
}
// => 0 1 2 3 4 5 6 7 8 9

// Exemple de parcours sans itérateur :
for (int i=0 ; i<vec.size() ; i++ ) System.out.println( vec.elementAt(i) ) ;

// Exemple de parcours avec itérateur :
for (Enumeration e = vec.elements() ; e.hasMoreElements() ; ) System.out.println(e.nextElement()) ;
```

### 9.4.3 La classe java.util.Hashtable

Supposons que l'on veuille chercher un objet dans un conteneur (par exemple l'adresse de quelqu'un dans un carnet d'adresses) : Avec un **Vector** la seule méthode à utiliser sera : je parcours mon conteneur tant que je ne trouve pas l'élément qui m'intéresse (tant que je n'ai pas trouvé l'adresse correspondant au nom de la personne qui m'intéresse)

La classe **Hashtable** permet de créer des conteneurs qui savent indexer chaque objet contenu par une clé unique : si je cherche un élément, il suffit de connaître sa clé (par exemple le nom de la personne dont on cherche l'adresse) et la table de hashing fournira un accès direct et performant à cet élément.

Les clés peuvent être n'importe quel objet implémentant la méthode **hashCode()** de la classe **Object.** C'est par exemple le cas des chaînes de caractères (String) ou des Wrapper. Une certaine clé ne peut identifier qu'une seule valeur.

Les conteneurs Hashtable sont ***très performants pour les opérations d'accès directs*** aux objets contenus.

Voici un échantillon des méthodes disponibles :

**Object put(Object cle, Object elt)**: Insère l'élément elt avec la clé cle
**Object remove(Object cle)** : Supprime et retourne l'objet correspondant à la clé cle
**Object get(Object cle)** : Retourne l'objet correspondant à la clé cle
**boolean containsKey(Object cle)** : Teste si un objet correspondant à la clé existe
**keys()** : Retourne une Enumeration sur les clés
**elements()** : Retourne une Enumeration sur les éléments

```
class Adresse {
        private String nom, adresse ;
        public Adresse(String nom, String adresse) { this.nom = nom ; this.adresse = adresse ; }
        public String toString() { return nom + " : " + adresse ; }
}

Hashtable carnet = new Hashtable() ;

carnet.put("Jean Dupond", new Adresse("Jean Dupond", "9, Rue Sans Nom 99000 Nullepart") );
carnet.put("Marc Durand", new Adresse("Marc Durand", "18, avenue Grande 98000 Ailleurs")) ;
…
Adresse rech = (Adresse)carnet.get("Jean Dupond") ;
```

***Remarque*** : La méthode ***get()*** retourne une référence du type **Object** => Nous sommes donc obligés de convertir explicitement la référence retournée en objet de la classe **Adresse**.

Dans l'exemple précédent, on peut considérer que le conteneur ***carnet*** va permettre de contenir l'ensemble des objets de la classe ***Adresse***. Il serait donc logique de considérer qu'il s'agit d'un attribut de cette classe et d'encapsuler ainsi la gestion des membres de cette classe (avec des méthodes static). Reprenons l'exemple :

```
class Adresse {
        private static Hashtable carnet = new Hashtable() ;
        private String nom, adresse ;

        public Adresse(String nom, String adresse) {
                this.nom = nom; this.adresse = adresse ;
                carnet.put(nom, this) ;
                }
        public String toString() { return nom + " : " + adresse ; }

        public static Adresse get( String nom ) { return (Adresse)carnet.get(nom) ; }
        public static Enumeration adresses() { return carnet.elements() ; }
}


new Adresse("Jean Dupond", "9, Rue Sans Nom 99000 Nullepart") ;
new Adresse("Marc Durand", "18, avenue Grande 98000 Ailleurs") ;
…
Adresse rech = Adresse.get("Jean Dupond") ;

for (Enumeration e = Adresse.adresses() ; e.hasMoreElements() ; )
        System.out.println( e.nextElement() ) ;          // inutile de convertir l'objet, car toString() est
                                                         // une méthode de la classe Object
```

# 10. L'interface homme/machine : AWT

## 10.1 Introduction

Le package **java.awt** contient les classes graphiques de base fournies par le JDK. Ces classes s'appuient sur des implémentations propres à chaque système (couche Peer). Une même application écrite avec des classes AWT, aura une apparence différente sur chaque système (elle ressemblera à une application Windows sur un système Windows, à une application X11 sur un système Unix, …).

Avec la version 1.2 du JDK, un nouveau package est apparu, le package **swing**, qui ne repose pas sur le système graphique de chaque système, mais au contraire propose des composants entièrement dessinés. Nous n'aborderons pas ce package, qui est similaire par son utilisation au package AWT.

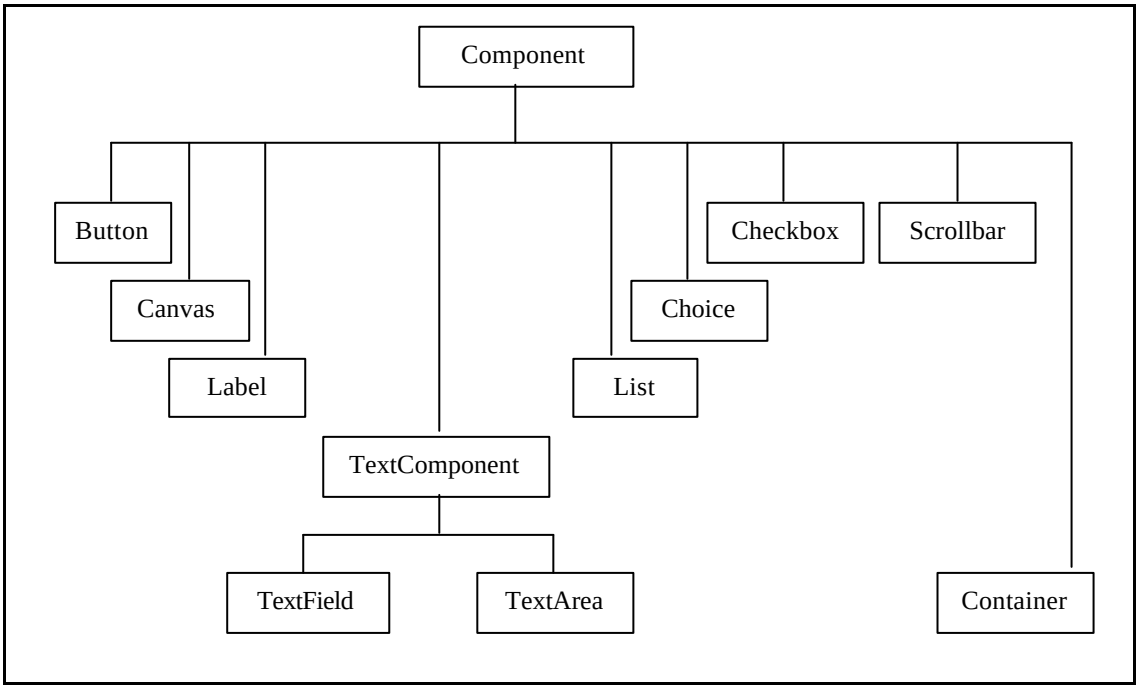
L'outil de conception que nous utilisons en TP (**Jbuilder 2** de ***Inprise***) permet de concevoir visuellement les interfaces graphiques et de générer automatiquement le code Java associé. Nous passerons donc rapidement sur l'étude de AWT, puisqu'il nous suffit de pouvoir comprendre le code généré automatiquement par notre outil de développement.

Une partie du package **swing** est d'ailleurs utilisable avec cet outil de développement….

## 10.2 Les composants : java.awt.Component

La classe **Component** est la classe de base de toute la hiérarchie des composants visibles de l'interface. Cette classe définit presque toutes les méthodes utiles (ces méthodes sont, bien entendu, souvent redéfinies par ses classes filles).

***Hiérarchie des composants :***



Description des différents types de composants :

| | |
|---|---|
| **Button** | : Représente un Bouton. |
| **Canvas** | : Composant graphique qui par défaut n'a pas de représentation visible et qui peut être utilisé pour dessiner quelque chose de spécifique. |
| **Label** | : Représente du texte affiché. |
| **TextField** | : Permet de représenter une ligne de texte éditable. |
| **TextArea** | : Permet de représenter une zone de texte éditable (plusieurs lignes). |
| **List** | : Représente une liste de choix sur une colonne. |
| **Choice** | : Représente une boite de choix (liste défilante). |
| **Checkbox** | : Représente les boutons radio et les cases à cocher. |
| **Scrollbar** | : Représente des ascenseurs. |
| **Container** | : La classe de base des classes permettant de représenter des composants pouvant en contenir d'autres. |

Voici quelques méthodes communes à tous les composants

- Géométrie du composant :

| | |
|---|---|
| **Rectangle getBounds()** | : Permet de récupérer la position et la dimension du composant. |
| **void setLocation(Point p)** | : Change la position du composant. |
| **void setSize(Dimension d)** | : Change la taille du composant. |
| **void setBounds(Rectangle r)** | : Change la taille et la position. |
| **Container getParent()** | : Retourne une référence sur le composant qui contient ce composant. |
| **Dimension getPreferredSize()** | : Retourne la "meilleure" dimension pour afficher le composant (tient compte de son contenu, de son environnement, …). |

Un objet de la classe **Rectangle** représente un rectangle. Il contient 4 attributs publics définissant sa position (x et y) et sa taille (width et height).

Un objet de la classe **Point** représente un point dans l'espace graphique. Il contient 2 attributs publics définissant sa position (x et y).

Un objet de la classe **Dimension** représente une largeur et une hauteur. Elle contient 2 attributs publics définissant cette taille (width et height).

Ces trois classes contiennent de nombreuses méthodes qu'il serait fastidieux de décrire ici.

- Comportement du composant :

| | |
|---|---|
| **void setVisible(boolean)** | : Affiche ou masque le composant. |
| **boolean isVisible()** | : Permet de savoir si le composant est visible. |
| **void requestFocus()** | : Permet à un composant de prendre la main (il devient le composant actif graphiquement). |
| **void setEnabled(boolean)** | : Active ou désactive le composant (Ex : bouton grisé ou non). |
| **boolean isEnabled()** | : Permet de savoir si un composant est actif. |

- Aspect du composant :

| | |
|---|---|
| **void setBackGround(Color c)** | : Couleur d'arrière plan. |
| **Color getBackGround()** | : Couleur d'arrière plan. |
| **void setForeGround(Color c)** | : Couleur d'avant plan. |
| **Color getForeGround()** | : Couleur d'avant plan. |

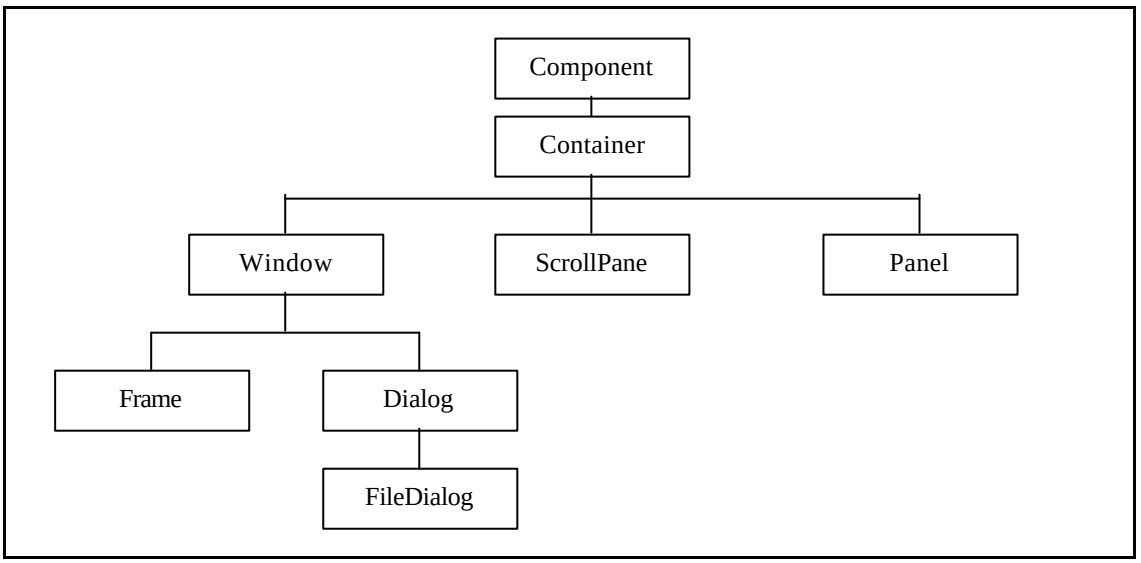
Un objet de la classe **Color** représente une couleur. Un objet de cette classe est construit en passant au constructeur trois valeurs comprises entre 0 et 255 et représentant le Rouge, le Vert et le Bleu (codage RGB, au total 16 millions de couleur). Plusieurs couleurs sont prédéfinies sous la forme d'attributs static de la classe : **Color.white**, **Color.black**, **Color.red**, …

La classe **Font** représente une police de caractères : nom de la police, taille et style utilisé (Gras, italiques, …).

## 10.3 Les conteneurs : java.awt.Container

La classe **Container**, sous-classe de la classe **Component**, représente un composant graphique capable de contenir plusieurs autres composants, y compris d'autres **Container**.

***Hiérarchie des conteneurs :***




Description des différents types de conteneurs :

| | |
|---|---|
| **Window** | : Représente une fenêtre vide sans rebord ni barre de titre. |
| **Frame** | : Représente une fenêtre possédant une barre de titre et dont l'apparence est proche des fenêtres "normales" de l'environnement d'exécution. |
| **Dialog** | : Représente une boite de dialogue qui peut être modale ou non |
| **FileDialog** | : Représente une boite d'ouverture ou d'enregistrement de fichiers |
| **Panel** | : Permet de représenter différents éléments graphiques en les liant entre eux dans une certaine politique de disposition. Nous reviendrons sur ce point dans le prochain chapitre. Toute fenêtre doit obligatoirement contenir au moins un Panel. |

***Remarque*** : Par défaut, une fenêtre (Windows ou Frame) est masquée et possède une taille nulle. Il faudra donc penser à la montrer (méthode setVisible(true)) et à changer sa taille (méthode setSize(…) ou pack()).

Un Container dispose de toutes les méthodes que nous avons vues dans le chapitre précédent (puisqu'il s'agit aussi d'un composant…)

Les méthodes propres aux conteneurs seront surtout liées à l'ajout et le retrait de composants dans ce conteneur.
Méthodes propres aux **Container** :

**Component add(Component c)** : Ajoute le composant c dans le conteneur.
**Component add(Component c, Object contrainte)** : Ajoute le composant c dans le conteneur en fonction de la contrainte fournie (qui dépend de la politique de disposition actuelle).
**void remove(Component c)** : Supprime le composant c du conteneur.

Méthodes propres à la classe **Frame** :

**void pack()** : Permet de calculer une taille idéale de la fenêtre en fonction de la taille de tous les composants qui se trouvent à l'intérieur et de la politique de disposition associée à ce Frame (nous reviendrons dessus).

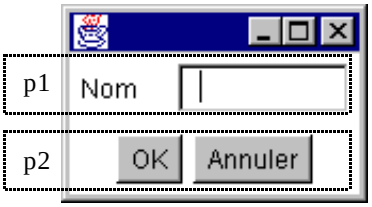
```
public class Panneaux extends Frame {

        Panneaux() {
                setLayout( new BorderLayout() ) ;  // spécifie la politique de disposition (par frontières)
                Panel p1 = new Panel() ;
                Panel p2 = new Panel() ;

                add(p1, "North" ) ;                    // Ajoute le composant p1 sur la frontière Nord
                add(p2, "South" ) ;                    // Ajoute le composant p2 sur la frontière Sud
                p1.add( new Label("Nom") ) ;
                p1.add( new TextField(8) ) ;
                p2.add( new Button("OK") ) ;
                p2.add( new Button("Annuler") ) ;
                pack() ;              // Calcul automatique de la taille de la fenêtre
        }

        public static void main(String args[] ) {
                Panneaux f  = new Panneaux() ;
                f.setVisible(true) ;
        }
}
```




## 10.4 Les gestionnaires d'aspect : java.awt.LayoutManager

### 10.4.1 Généralités

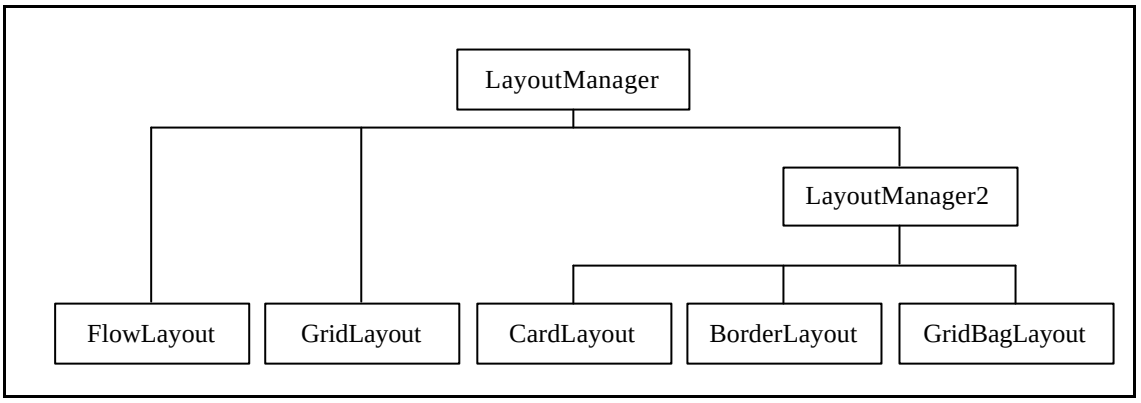
Ces classes, qui héritent toutes de LayoutManager, définissent différentes politiques de disposition des composants dans un conteneur.
Une certaine politique permet de positionner les composants et de contrôler leur dimension.
Nous présenterons quelques gestionnaires fournis avec AWT, mais il est tout à fait possible de définir ses propres gestionnaires ou d'en récupérer sur Internet…

Pour associer un gestionnaire d'aspect à un conteneur, il suffit d'utiliser la méthode de conteneur suivante:

**void setLayout(LayoutManager l)**

Si aucun gestionnaire n'est associé à un conteneur, le positionnement des composants doit être assuré entièrement par l'application (ce qui est laborieux).

Voici les gestionnaires fournis avec AWT :




## 10.4.2 Java.awt.FlowLayout

C'est le gestionnaire d'aspect par défaut des Panel et des Applet.

Son principe est simple : les composants sont placés les uns à coté des autres.

```
public class Panneaux2 extends Frame {

        Panneaux2() {
                setLayout( new FlowLayout() ) ;     // spécifie la politique de disposition

                add( new Label("Nom") ) ;
                add( new TextField(8) ) ;
                add( new Button("OK") ) ;
                add( new Button("Annuler") ) ;
                pack() ;              // Calcul automatique de la taille de la fenêtre
        }

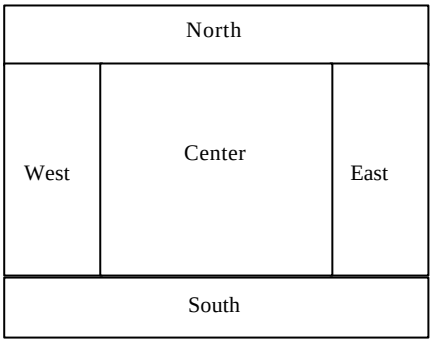
        public static void main(String args[] ) {
                Panneaux2 f  = new Panneaux2() ;
                f.setVisible(true) ;
        }
}
```



### 10.4.3 Java.awt.BorderLayout

C'est le gestionnaire d'aspect par défaut des Frame.

Il permet de décomposer la zone graphique du conteneur en 5 zones :



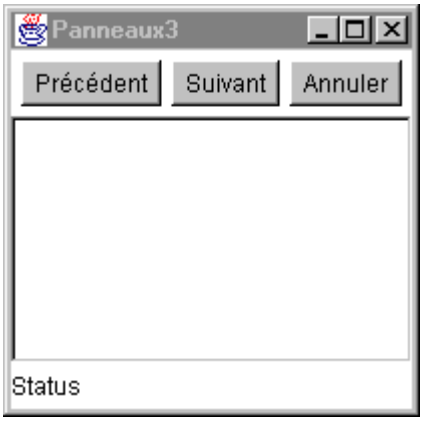

On ne peut placer qu'un seul composant au maximum par zone, mais pour en mettre plus il suffit que ce composant soit lui-même un Panel dans lequel on pourra à nouveau placer des composants.

```
public class Panneaux3 extends Frame {

        Panneaux3() {
                super("Panneaux3");
                setLayout( new BorderLayout() ) ;  // spécifie la politique de disposition (par frontières)
                Panel p = new Panel() ;

                add(p, "North" ) ;              // Ajoute le composant p sur la frontière Nord
                p.add( new Button("Précédent") ) ;
                p.add( new Button("Suivant") ) ;
                p.add( new Button("Annuler") ) ;
                add(new ScrollPane(), "Center" ) ;   // Ajoute le composant au centre
                add(new Label("Status"), "South" ) ;// Ajoute le composant sur la frontière Sud
                pack() ;              // Calcul automatique de la taille de la fenêtre
        }

        public static void main(String args[] ) {
                Panneaux3 f  = new Panneaux3() ;
                f.setVisible(true) ;
        }
}
```




## 10.5 La gestion des graphiques : java.awt.Graphics

Chaque composant est associé à un objet Graphics qui est sa représentation graphique et permet de dessiner dans ce composant.
On ne peut effectivement modifier la représentation graphique que dans les composants suivants : **Canvas**, **Panel**, **Window**, **Frame**, **ScrollPane**.

Le dessin d'un composant doit se trouver dans la méthode:

void **paint**(Graphics g)

Cette méthode est appelée automatiquement par la plate-forme graphique lorsque cela est nécessaire (c'est à dire lorsque le composant doit être dessiné).
Pour réaliser des dessins spécifiques, on redéfinit généralement cette méthode dans une sous-classe de Canvas.

De nombreuses méthodes permettant de dessiner sont proposées par la classe **Graphics** : ***drawRect(), drawLine(), drawPolygon(), drawArc(), drawString(), fillRect(), fillPolygon()***, …

On ne doit jamais appeler la méthode **paint()** directement : pour demander à un composant de se redessiner, il faut toujours appeler la méthode **repaint()**. Cette méthode, qui ne doit pas être redéfinie, permet de demander à un Thread de la machine Virtuelle Java, le "Screen Updater" de redessiner le composant au bon moment (c'est à dire lorsqu'il n'y a rien de plus urgent à faire). La méthode **paint()** sera alors appelée de manière asynchrone…

## 10.6 La gestion des événements

### 10.6.1 Principes

Dans tout système de gestion d'un environnement graphique il est nécessaire de pouvoir se mettre à l'écoute d'événements (comme le clic sur un bouton) et de savoir exécuter du code pour chaque événement ainsi détecté. Java dispose de son propre système d'événements qui permet de capter des événements liés à l'interface graphique AWT ou à d'autres origines. Il est d'ailleurs tout à fait possible de générer ses propres événements : les événements constituent ainsi un système de communication entre les différents composants d'un programme Java et avec son environnement externe.

Le modèle utilisé dans Java que nous allons décrire ici n'est disponible que depuis la version 1.1 du JDK.

Ce modèle est basé sur la délégation. On distingue :

- Des **sources** d'événements (n'importe quel Composant de AWT est par exemple une source d'événement).
- Les événements eux-mêmes qui dérivent de la classe **EventObject**.
- Des récepteurs d'événements (appelé **Listener**).

Le traitement des événements se décompose ainsi :

1. Un Listener s'abonne auprès d'une source pour un certain type d'événement. Plusieurs Listener peuvent ainsi s'abonner sur le même événement dans la même source.
2. L'utilisateur réalise une action qui provoque l'apparition de l'événement dans la source d'événement (par exemple un clic de souris).
3. L'événement est diffusé vers tous les Listener qui sont abonnés.
4. Les Listener concernés traitent chacun à leur manière cet événement (ils peuvent notamment être la source d'un nouvel événement).

***Remarque*** : Dans un outil graphique tel que Jbuilder, la gestion des événements dans le programme est largement automatisée : lorsque l'on ajoute un composant graphique au programme, une fenêtre permet de voir la liste de tous les événements que ce composant peut provoquer. Il suffit de double-cliquer sur l'un de ces événements pour que tout le code nécessaire à la récupération de cet événement soit généré automatiquement (création du Listener et abonnement) ; il ne reste plus qu'à écrire le code qui doit être exécuté à l'apparition de cet événement.

### 10.6.2 Les sources d'événements

N'importe quel objet peut potentiellement être une source d'événement. Dans AWT tous les Composants (Component) sont des sources d'événements.

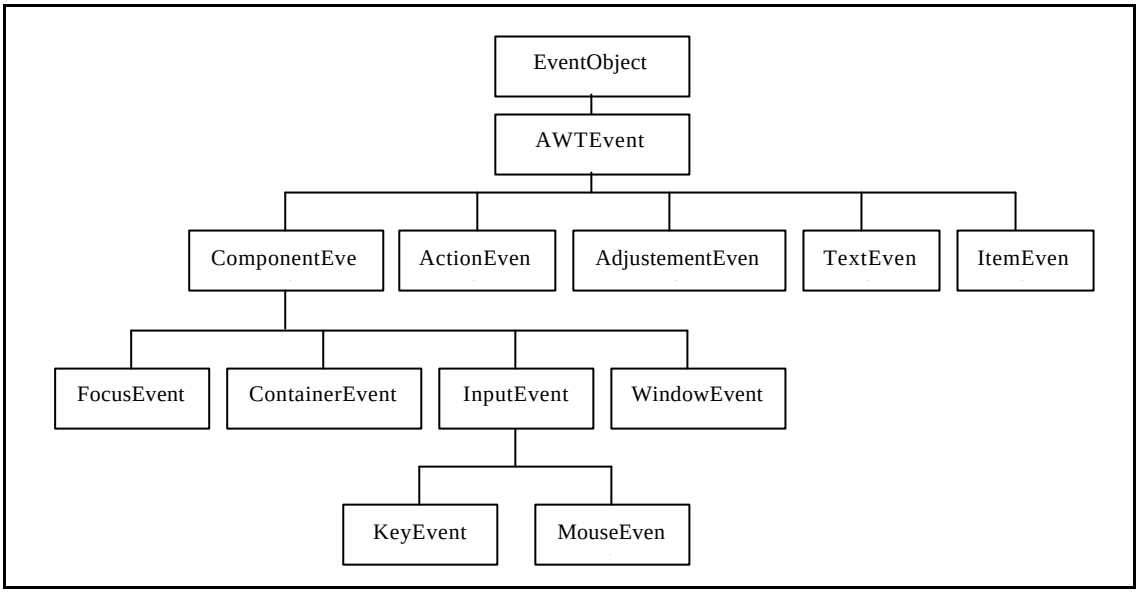
Un Listener peut s'abonner auprès d'une source avec la méthode :

**void addXxxxListener(Listener l)** : où Xxxx est l'événement concerné

```
Button b = new Button("OK") ;
Listener listener = new …
b.addActionListener(listener) ;      // L'objet listener est abonné sur l'événement Action de b
                                     // (cet événement est déclenché lorsque l'on clic ce bouton)
```

### 10.6.3 Les événements de l'AWT

Voici une liste partielle des événements générés par AWT :




**ComponentEvent** : Représente tous les événements pouvant être associés à un composant (ces événements sont généralement redirigés vers le gestionnaire d'aspect associé au conteneur de niveau supérieur).
**FocusEvent** : Evénements liés au gain ou à la perte de focus.
**ContainerEvent** : Evénements liés aux conteneurs (ajout, suppression de composants, …).
**WindowEvent** : Evénements liés à la classe Window (activation, ouverture, …).
**InputEvent** : Evénements générés par un périphérique d'entrée.
**KeyEvent** : Evénements générés par le clavier.
**MouseEvent** : Evénements générés par la souris.
**ActionEvent** : représente la réalisation d'une action typique du composant associé (clic sur un bouton, séléction dans un menu, …). C'est ce type d'événement qui est le plus souvent utile.
**AdjustementEvent** : Indique le changement d'une valeur numérique (par exemple le niveau d'un ascenseur).
**TextEvent** : La valeur d'un champ texte a été modifiée.
**ItemEvent** : Changement de l'état d'une sélection (Checkbox, Choice, List, …).

### 10.6.4 Listener d'événements

Un Listener est un objet qui doit implémenter une interface qui dérive elle-même de l'interface EventListener. Il existe une interface Listener par type d'événement :

Interface **XxxxListener** si l'événement est Xxxx

Chacune de ces interfaces contient une ou plusieurs méthodes correspondant aux différents événements pouvant être reçus : lorsque tel événement sera émis par la source d'événement, telle méthode sera appelée automatiquement sur les Listener abonnés à cette source.

```
class Coucou implements ActionListener {
        public void actionPerformed(ActionEvent e ) {
                System.out.println("Coucou") ;
        }
}

Button b = new Button("Affiche Coucou") ;
b.addActionListener( new Coucou() ) ;
```

On voit que les Listener sont souvent de très petites classes, utilisées une seule fois. Il est donc préférable d'utiliser des classes anonymes :

```
Button b = new Button("Affiche Coucou") ;
b.addActionListener( new ActionListener() {
                        public void actionPerformed(ActionEvent e ) {
                                System.out.println("Coucou") ;
                                }
                        } ) ;
```

# 11. Les entrées/sorties

## 11.1 Généralités

Ce chapitre est consacré aux classes permettant de réaliser des entrées/sorties (typiquement pour lire et écrire dans un fichier ou plus généralement dans toute source ou destination de données).

Il existe dans Java deux groupes de classes :

- **Les entrées/sorties par octet** ou manipulations de Stream (flux d'octets)
- **Les entrées/sorties par caractère** UNICODE ou utilisation de Reader/Writer (flux de caractères)

Les Reader/Writer ont étés introduits avec le JDK 1.1 pour permettre la manipulation de caractères internationaux (UNICODE).

Les classes **InputStream** et **OutputStream** sont les classes de base des flux d'octets

Les classes **Reader** et **Writer** sont les classes de base des flux de caractères.

Les classes **InputStreamReader** et **OutputStreamWriter** permettent de faire le lien entre les deux modèles ( en permettant de transformer un flux d'octets en flux de caractères).

Les classes d'entrées/sorties se décomposent en deux types de classes :

- Les classes de **type canal** qui représentent des **sources des données** ou plus exactement qui permettent de se brancher à de telles sources (fichiers, tableaux en mémoire, sockets). Ces classes ***dérivent directement des classes InputStream/OutputStream et Reader/Writer***.
- Les classes de **type Filtre** qui peuvent **se connecter sur un Canal ou sur un autre Filtre** et permettent d'interpréter les octets ou les caractères reçus à un haut niveau (lire/écrire un entier, un flottant, …). Ces classes ***dérivent des classes FilterInputStream/ FilterOuputStream et FilterReader/FilterWriter***.

Réaliser des entrées/sorties ressemble beaucoup à un jeu de mécano : on crée un Canal et on branche dessus un ou plusieurs Filtres pour obtenir les données dans le format qui nous intéresse…

Voici quelques exemples de classes de type Canal :

**FileInputStream**/**FileOutputStream** : lire/écrire dans un fichier en mode octet
**FileReader**/**FileWriter** : lire/écrire dans un fichier en mode caractère

**ByteArrayInputStream**/**ByteArrayOutputStream** : dans un tableau d'octets en mode octet
**CharArrayReader**/**CharArrayWriter** : dans un tableau de caractères en mode caractère

Et quelques exemples de classes de classes de type Filtre :

**DataInputStream**/**DataOutputStream** : lire/écrire des données formatées (int, float, …)

**BufferedInputStream**/**BufferedOutputStream** : E/S bufferisées de ou vers un Stream
**BufferedReader**/**BufferedWriter** : E/S bufferisées depuis un Reader ou vers un Writer
**PrintWriter** : Très adapté à l'écriture de données formatées vers un Writer (méthode println)

## 11.2 La classe java.io.File

Cette classe permet de manipuler le système de fichier. Elle propose en particulier les méthodes :

**public File(File dir, String nom) ;**
**public File(String path) ;**
**boolean isDirectory() ;**
**boolean isFile()**
**String getAbsolutePath() ;**
**void renameTo(String dest) ;**
**boolean mkdir(String nom) ;**
**String list() ;** // retourne la liste des fichiers contenus dans un répertoire
**int length() ;**
**boolean delete() ;**
**boolean canRead() ;**
**boolean canWrite() ;**

```
import java.io.* ;

public class Dir {
	public static void main(String[] args) {
		if (args.length != 1) {
			System.out.println("Utilisation : Dir répertoire") ;
			System.exit(0) ;
		}
		try {
			File dir = new File(args[0]) ;
			File fich = null ;
			String[] liste = dir.list() ;
			for(int i=0 ; i<liste.length ; i++) {
				System.out.print(liste[i]) ;
				if ( fich = new File(args[0], liste[i]).isDirectory())
					System.out.println("\tDIR") ;
				else	System.out.println("\t" + fich.length()) ;
			}
		}
		catch (Exception e ) {
			System.out.println(e) ;
		}
	}
}
```

## 11.3 Exemples d'entrées/sorties par octet

### 11.3.1 Utilisation des canaux FileInputStream et FileOutputStream

Ces deux classes représentent des canaux d'octets associés à la lecture et l'écriture dans des fichiers.

Méthodes de la classe **FileInputStream** :

**FileInputStream(File file) ;**
**FileInputStream(String nom) ;** // construction à partir du nom de fichier
**void close() ;**
**int available() ;** // retourne le nombre d'octets lisibles sans bloquer le flux
**int read() ;** // lit un octet
**int read(byte[]) ;** // lit un tableau d'octet
**int read(byte[], int indiceDépart, int taille) ;**

Méthodes de la classe **FileOutputStream** :

**FileOutputStream(File file) ;**
**FileOutputStream(String nom) ;**
**void close() ;**
**int write() ;**
**int write(byte[]) ;**
**int write(byte[], int indiceDépart, int taille) ;**

***Remarque*** **:** Sauf pour les constructeurs (qui dépendent de la nature du canal), ces méthodes sont communes à toutes les classes de type canal.

```
import java.io.* ;

public class CopieFichier {
        public static void main(String args[]) {
                if (args.length != 2) {
                    System.out.println("Utilisation : CopieFichier source destination") ;
                    System.exit(1) ;
                }
                try { // Ouverture des fichiers et mise en place des filtres :
                        FileInputStream fin = new FileInputStream(args[0]);
                        FileOutputStream fout = new FileOutputStream(args[1]);
                }
                catch (IOException e) {
                        System.out.println("Impossible d'ouvrir les fichiers : " + e) ;
                        System.exit(1) ;
                }
                try { // Copie des fichiers et fermeture
                        byte[] memoire = new byte[fin.available()] ;
                        fin.read( memoire, 0, memoire.length ) ;
                        fout.write( memoire, 0, memoire.length ) ;
                        fin.close() ;
                        fout.close() ;
                }
                catch (IOException e) {
                        System.out.println("Impossible de copier les fichiers : " + e) ;
                        System.exit(1) ;
                }
        }
```

## 11.3.2 Utilisation des filtres BufferedInputStream et BufferedOutputStream

Ces classes représentent des filtres permettant de bufferiser des canaux, afin d'améliorer les performances de lecture ou d'écriture sur ces canaux. Elles offrent des méthodes proches de celles des canaux de base.

Méthodes de la classe **BufferedInputStream** :

**BufferedInputStream(InputStream canal) ;** // Construction à partir d'un canal
**void close() ;** // permet de fermer le canal associé
**int available() ;** // retourne le nombre d'octets lisibles sans bloquer le flux
**int read() ;** // lit un octet
**int read(byte[]) ;** // lit un tableau d'octet
**int read(byte[], int indiceDépart, int taille) ;**

Méthodes de la classe **BufferedOutputStream** :

**BufferedOutputStream(OutputStream canal) ;**

**void close() ;**
**int write() ;**
**int write(byte[]) ;**
**int write(byte[], int indiceDépart, int taille) ;**

Dans l'exemple ci-dessous des filtres de type **Buffered** sont utilisés pour améliorer les performances de canaux sur des fichiers :

```
import java.io.* ;

public class CopieFichierAvecBuffer {
        public static void main(String args[]) {
                if (args.length != 2) {
                   System.out.println("Utilisation : CopieFichierAvecBuffer source destination") ;
                   System.exit(1) ;
                        }
                try { // Ouverture des fichiers et mise en place des filtres :
                        BufferedInputStream bin =
                                new BufferedInputStream(new FileInputStream(args[0]));
                        BufferedOutputStream bout =
                                new BufferedOutputStream(new FileOutputStream(args[1]));
                }
                catch (IOException e) {
                        System.out.println("Impossible d'ouvrir les fichiers : " + e) ;
                        System.exit(1) ;
                }
                try { // Copie des fichiers et fermeture
                        byte[] memoire = new byte[bin.available()] ;
                        bin.read( memoire, 0, memoire.length ) ;
                        bout.write( memoire, 0, memoire.length ) ;
                        bin.close() ;
                        bout.close() ;
                }
                catch (IOException e) {
                        System.out.println("Impossible de copier les fichiers : " + e) ;
                        System.exit(1) ;
                }
        }
```

### 11.3.3 Utilisation des Filtres DataInputStream et DataOutputStream

Ces classes sont des filtres qui permettent de lire ou écrire des types de base Java sur un canal d'octet.

Exemples de méthodes de la classe **DataInputStream** :

**DataInputStream(InputStream canal) ;**
**int readInt() ;**
**float readFloat() ;**
**char readChar() ;**
**…**
**void readFully( byte b[] ) ;** // lire tous les octets dans b

Exemples de méthodes de la classe **DataOutputStream** :

**DataOutputStream(OutputStream canal) ;**
**void writeInt(int) ;**
**void writeFloat(float) ;**
**void writeChar(char) ;**

…
**void write( byte b[] ) ;**

```
// Exemple de deux méthodes permettant d'écrire et lire un tableau de int
public void write(String file, int tab[]) throws IOException {
        DataOutputStream out = new DataOutputStream(new FileOutputStream(file)) ;
        out.writeInt(tab.length) ;
        for(int i=0 ; i<tab.length ; i++) out.writeInt(tab[i]) ;
        out.close() ;
}
public int[] read(String file) throws IOException {
        DataInputStream in = new DataInputStream(new FileInputStream(file)) ;
        int lg = in.readInt() ;
        int[] tab = new int[lg] ;
        for (int i=0 ; i<lg ; i++) tab[i] = in.readInt() ;
        in.close() ;
}
```

## 11.4 Exemples d'entrées/sorties par caractère

### 11.4.1 Utilisation des canaux FileReader et FileWriter

Ils s'utilisent de la même manière que **FileInputStream** et **FileOutputStream**…

### 11.4.2 Utilisation des filtres BufferedReader et BufferedWriter

Ils s'utilisent de la même manière que **BufferedInputStream** et **BufferedOutputStream**…

Ils apportent de plus des méthodes du type :

**String readLine() ;** pour BufferedReader
**void write(String ligne) ;** pour BufferedWriter

### 11.4.3 Utilisation du filtre PrintWriter

Cette classe est un filtre qui permet d'écrire les types de base, des chaînes de caractères et tout type ou objet disposant de la méthode ***toString()*** vers un canal de sortie en mode caractère : pour écrire ces données ; on utilise les méthodes ***print()*** et ***println()***.

```
try {
        FileWriter fout = new FileWriter("fich.txt") ;
        PrintWriter pout = new PrintWriter(fout) ;
        pout.print("sans retour à la ligne");
        pout.println("avec retour à la ligne");
}
```

### 11.4.4 Utilisation de OutputStreamWriter et InputStreamReader

Ces deux classes permettent de transformer des flux d'octets en flux de caractères.

```
// L'objet System.in est de type InputStream
        …
        BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in)) ;
        System.out.print("Saisir une ligne : ");
        System.out.println( stdin.readLine() ) ;
}
```

# 12. Les threads

## 12.1 Principes généraux

Dans un système classique (multitâche), une tâche est un objet disposant de son propre code et surtout de ses propres données ce qui se révèle lourd à gérer (il est très coûteux de réserver et d'initialiser de la mémoire à chaque création de tâche).
Un thread est une tâche légère qui partage sa mémoire avec les autres threads de la même application. Un thread peut donc être créé très rapidement.

Le langage Java permet de gérer facilement des Threads et donc permet de réaliser une programmation multitâche souple et légère.

Les Threads Java s'appuient sur les threads natifs des systèmes d'exploitation (Windows 95, Windows NT, Unix, …). La machine virtuelle Java elle-même utilise des Threads : par exemple le Garbage Collector est un Thread qui s'exécute en permanence en tâche de fond.

Il existe deux méthodes dans Java pour créer une classe qui représente un thread :

1. en héritant de la classe **Thread**.
2. en implémentant l'interface **Runnable**.

## 12.2 Mise en œuvre avec la classe java.lang.Thread

La classe **Thread** encapsule les opérations systèmes permettant de créer, exécuter et synchroniser les threads. Elle masque les différences d'implémentation des threads d'un système à un autre.

Voici quelques méthodes de la classe Thread :

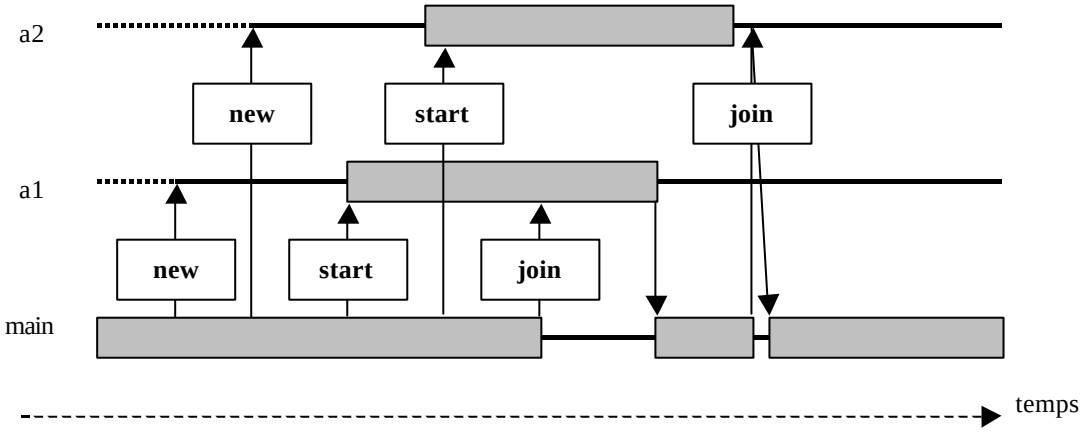
**void start() ;** // Permet de demander au Thread concerné de démarrer (asynchrone : cette fonction redonne immédiatement la main pour la suite des instructions).
**void stop() ;** // Permet de demander au Thread concerné de s'arrêter.
**void join() ;** // On attend que le Thread concerné se soit terminé.
**void run() ;** // Méthode à redéfinir pour décrire les actions d'un Thread (c'est le contenu de cette méthode qui représente les actions du thread et qui sera donc exécuté après que le Thread aura été démarré par la méthode start().).

```
Class ActionDeCuisine extends Thread {
        private String afaire ;
        public ActionDeCuisine(String afaire) { this.afaire = afaire ; }

        public void run() { // le contenu du Thread
                for (int i=0 ; i<4 ; i++) System.out.println(afaire) ;
        }
}
class Recette {
        public static void main(String[] args) {
                ActionDeCuisine a1 = new ActionDeCuisine("Battre les œufs");
                ActionDeCuisine a2 = new ActionDeCuisine("Faire fondre le Chocolat");

                a1.start() ;
                a2.start() ;
                a1.join() ;
                a2.join() ;
                System.out.println("Mélanger et Cuire") ;
        }
}
```

Représentation du fonctionnement de ce programme :




On voit que les threads g1 et g2 ne sont actifs qu'une fois démarrés, jusqu'à la fin de leur méthode *run()*. Le thread ***main*** est actif pour la durée de l'application (en dehors des périodes où il attend d'autres threads avec *join()*).

Le résultat exact de cette application dépend du système d'exploitation. Résultats possibles :

```
Battre les œufs
Faire fondre le Chocolat
Battre les œufs
Faire fondre le Chocolat
Battre les œufs
Faire fondre le Chocolat
Battre les œufs
Faire fondre le Chocolat
Mélanger et Cuire
```

```
Battre les œufs
Battre les œufs
Faire fondre le Chocolat
Faire fondre le Chocolat
Battre les œufs
Battre les œufs
Faire fondre le Chocolat
Faire fondre le Chocolat
Mélanger et Cuire
```

```
Battre les œufs
Battre les œufs
Battre les œufs
Battre les œufs
Faire fondre le Chocolat
Faire fondre le Chocolat
Faire fondre le Chocolat
Faire fondre le Chocolat
Mélanger et Cuire
```

***Explications*** : Un système d'exploitation multi-thread affecte des tranches de temps à chaque thread activable : il interrompt le thread actif quand une tranche de temps est écoulée et passe le contrôle au thread activable suivant. Le thread initial ne reprendra donc la main que lorsque son tour reviendra, après que les autres threads activables auront épuisé leur tranche de temps.
La durée des tranches de temps peut être très différente d'une machine à une autre, ce qui explique les différences de comportement d'un système à un autre.

## 12.3 Mise en œuvre avec l'interface java.lang.Runnable

La solution de mise en œuvre précédente a un grand défaut : une classe héritant de Thread ne peut hériter de rien d'autre (puisqu'il n'y pas d'héritage multiple). Cela peut parfois être très gênant.

Il existe une deuxième mise en œuvre possible des threads : On crée directement une instance de la classe **Thread** en passant un objet implémentant l'interface **Runnable** à son constructeur. L'objet **Thread** ainsi créé peut être utilisé exactement comme dans la mise en œuvre précédente. La classe qui contient le code a exécuter en parallèle doit simplement implémenter l'interface **Runnable** (Cette interface ne contient que la méthode ***run()*** qui sera appelée lors du démarrage du thread).

Reprise de l'exemple précédent :

```
Class ActionDeCuisine implements Runnable {
        private String afaire ;
        public ActionDeCuisine(String afaire) { this.afaire = afaire ; }

        public void run() { // le contenu du Thread
                for (int i=0 ; i<4 ; i++) System.out.println(afaire) ;
        }
}
class Recette {
        public static void main(String[] args) {
                Thread a1 = new Thread(new ActionDeCuisine("Battre les œfs")) ;
                Thread a2 = new Thread(new ActionDeCuisine("Faire fondre le Chocolat")) ;

                a1.start() ;
                a2.start() ;
                a1.join() ;
                a2.join() ;
                System.out.println("Mélanger et Cuire") ;
        }
}
```

***Remarque*** : Il est parfois utile dans la méthode ***run()*** d'un objet **Runnable** de pouvoir récupérer une référence sur le thread dans lequel cette méthode est exécutée. Pour cela il est possible d'utiliser la méthode :

```
static Thread currentThread() ;
```

```
public void run() {
        …
        Thread current = Thread.currentThread() ;
        …
}
```

## 12.4 Partage du temps

Deux méthodes de la classe Thread permettent à un Thread de passer volontairement la main aux autres Thread du système. Cela peut être utile, sur certains systèmes, si un thread risque de prendre tout le temps CPU. Cela peut aussi être un moyen de garantir que l'application aura le même comportement sur tous les systèmes.

Ces deux méthodes sont :

| | |
|---|---|
| **static void yield() ;** | // Permet au Thread courant de passer immédiatement la main aux autres (sans attendre la fin de sa tranche de temps et en espérant que la politesse lui sera rendue…) |
| **static void sleep(int ms)** | // L'activité du Thread courant est stoppée pendant le nombre de millisecondes indiqué |

```
public void run() { // le contenu du Thread
        for (int i=0 ; i<4 ; i++) {
                System.out.println(afaire) ;
                Thread.yield() ;
        }
}
```

```
public void run() { // le contenu du Thread
        for (int i=0 ; i<4 ; i++) {
                System.out.println(afaire) ;
                try { Thread.sleep(100) ; } catch (InterruptedException e) { }
        }
}
```

## 12.5 Partage de données

Tous les threads partagent le même espace d'adressage et peuvent donc entrer en conflit pour l'accès à une ressource.

Supposons que deux threads distincts (T1 et T2) réalisent au même moment l'instruction ***i = i+1***. Cette instruction consiste en fait à au moins trois instructions de base : 1) Lecture du contenu de i dans un registre 2) on ajoute 1 à ce registre 3) on stocke le résultat dans i.
Il pourrait donc se produire ceci (on suppose que i contient initialement la valeur 2 ) :

1. T1 lit i dans son registre (2)
2. T1 additionne 1 à son registre (3)
3. T2 lit i dans son registre (2)
4. T1 écrit son registre dans i =>3
5. T2 additionne 1 à son registre (3)
6. T2 écrit son registre dans i =>3

Au final i contient 3 au lieu de 4…

Il est donc nécessaire de disposer du moyen de dire "Personne ne peut faire cette instruction en même temps que moi, les autres doivent attendre que j'ai fini".

Dans Java, un **moniteur** peut être associé à chaque objet. Dés qu'un thread détient ce moniteur, les autres voulant accéder à ce moniteur sont mis en attente, et ne peuvent ni inspecter, ni modifier cet objet. Les segments de code qui permettent l'accès aux mêmes données depuis des threads séparés et qu'il faut protéger sont appelés "**sections critiques**". Ces sections doivent être marquées par le mot-clé **synchronized** pour être protégées par le moniteur de l'objet correspondant.

Il est possible de marquer des méthodes ou de simples blocs de code comme étant **synchronized**. Dés qu'un thread utilise une zone **synchronized** d'un certain objet, toutes les zones critiques de l'objet sont verrouillées : tout autre thread voulant accéder à une de ces zones sera mis en attente.

```
public class Point {
        private int x, y ;
        public synchronized void setXY(int x, int y) { this.x = x ; this.y = y ; }
        public synchronized void translate(int dx, int dy) { x += dx ; y += dy ; }
        public synchronized int getX() { return x ; }
        public synchronized int getY() { return y ; }
 }
```

```
public class Personne {
        private String nom ;
        void changeNom(String nom) {
                System.out.println("début changement") ;
                Synchronized (nom) {
                        this.nom = nom ;
                }
                System.out.println("fin changement");
        }
}
```

# 13. Le réseau

Ce chapitre concerne les classes du package **java.net** qui permettent de travailler avec le réseau au niveau applicatif. Il suppose de connaître les bases du protocole TCP/IP.

## 13.1 Les URL

### 13.1.1 Définitions

***URL = Uniform Resource Locator***

Une URL est une adresse qui permet de localiser une ressource sur un réseau.

Elle contient :

- Le protocole utilisé : http, ftp, file…
- Le nom du serveur ou son adresse IP
- Le chemin et le nom du fichier qui fournira les informations

http://www.atos-group.com/index.html
ftp://www.ibp.fr/pub/win3/outils/util.exe

Les classes Java ne supportent que les protocoles http, ftp et file.

### 13.1.2 Les classes URL et URLConnection

La classe **URL** permet de représenter une adresse. Cette classe fournit la méthode ***openConnection()*** qui permet d'ouvrir une connexion réseau correspondant à cette adresse. Cette méthode retourne un objet de type **URLConnection**.

La classe **URLConnection** permet de représenter une connexion à une ressource distante : elle permet de récupérer un flux d'octet en entrée ou en sortie vers cette ressource. (méthodes ***getInputStream()*** et ***getOutputStream()***).

```
class WebTexte {
        public static void main(String args[]) {
        try {
                URL u = new URL(args[0]) ;
                URLConnection c = u.openConnection() ;

                BufferedReader in =
                                new BufferedReader(new InputStreamReader(c.getInputStream()));
                String inLine ;
                while((inLine = in.readLine()) != null) System.out.println(inLine) ;
        }
        catch (IOException e) {
                System.out.println(e) ;
        }
        }
}
```

## 13.2 Les sockets

### 13.2.1 Présentation

Les sockets sont un ensemble de classes permettant de travailler directement au niveau des protocoles TCP/IP ou UDP/IP. L'utilisation de ces classes est proche de celle de la bibliothèque de fonctions C correspondantes.

Ces classes permettent de définir son propre protocole de communication applicatif pour faire communiquer deux applications sur le réseau.

Les classes à utiliser sont :

**ServerSocket** : pour se mettre à l'écoute d'une nouvelle connexion (coté serveur).
**Socket** : pour établir et gérer une connexion TCP (coté serveur et client).
**InetAddress** : pour construire une adresse réseau (coté client).
**DatagramSocket** : pour gérer une connexion UDP.

Nous n'aborderons dans ce cours que la programmation du protocole TCP/IP (nous ne verrons donc pas la classe **DatagramSocket**)

### 13.2.2 Le client : classes InetAddress et Socket

La classe **InetAddress** permet de construire et de représenter une adresse IP.

On ne peut pas directement construire un objet de cette classe, mais celle-ci offre plusieurs méthodes statiques qui permettent de construire et récupérer la référence d'un objet **InetAddress** :

```
static Inetaddress getByName(String nom) ;      // à partir d'un nom de serveur
static InetAddress getLocalHost() ;             // Pour l'adresse de la machine locale
```

```
InetAddress a1 = InetAddress.getByName("www.inria.fr") ;
InetAddress a2 = InetAddress.getByName("194.89.34.2") ;
InetAddress a3 = InetAddress.getLocalHost() ;
```

La classe **Socket** permet de créer et gérer une connexion TCP/IP vers un serveur distant sur un port donné. Voici quelques méthodes :

```
Socket(InetAddress adresse, int port) ;
InputStream getInputStream() ;
OutputStream getOutputStream() ;
void close();
int getPort() ;                 // retourne le port utilisé à distance
int getLocalPort() ;            // retourne le port utilisé localement
InetAddress getInetAddress() ;  // retourne l'adresse utilisée à distance
InetAddress getLocalAddress() ; // retourne l'adresse utilisée localement
```

```
import java.io.*
import java.net.*

public class Client {
      private InetAddress adresse ;
      private Socket sock ;

      Client(String serveur) { try {
            adresse = InetAddress.getByName(serveur) ;
            sock = new Socket(adresse, 5000) ;
            BufferedReader in = new BufferedReader(new InputStreamReader(sock.getInputStream())) ;
            PrintWriter out = new PrintWriter( new OutputStreamWriter(sock.getOutputStream()),true);
                                                                  // true permet de faire de l'auto-flush.
            out.println("Bonjour Serveur de la part de : " + InetAddress.getLocalHost()) ;
            System.out.println(in.readLine());
            sock.close() ;
      } catch (Exception e) { System.out.println(e) ; }
      }

      public static void main(String argv[]) { new Client( argv[0]) ; }
}
```

### 13.2.3 Le serveur : classes ServerSocket et Socket

Pour écrire un serveur, il est nécessaire de pouvoir se mettre à l'écoute de demandes de connexion en provenance des clients.
C'est le rôle des instances de la classe **ServerSocket** : on construit une instance de cette classe en lui fournissant le numéro du port sur lequel elle doit se mettre à l'écoute et il suffit alors d'utiliser la méthode ***accept()*** sur l'objet construit pour se mettre en attente d'une connexion.
La méthode ***accept()*** est bloquante jusqu'à ce qu'une connexion se produise et elle renvoie alors une référence sur un objet de type **Socket** pour gérer la nouvelle connexion.
La suite du code est alors similaire à ce qui se passe sur un client (utilisation de **Socket**).

```
import java.io.*
import java.net.*

public class Serveur {
   private ServerSocket sock ;
   private Socket client ;

   Serveur() { try {
      sock = new ServerSocket(5000) ;
      while(true) {
           client = sock.accept() ;
           BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream())) ;
           PrintWriter out = new PrintWriter( new OutputStreamWriter(client.getOutputStream()),true);
                                                                  // true permet de faire de l'auto-flush.
           System.out.println(in.readLine());
           out.println("Bonjour Client de la part de " + InetAddress.getLocalHost() ) ;
           client.close() ;
           }
      } catch (Exception e) { System.out.println(e) ; }
      }
   public static void main(String argv[]) { new Serveur( ) ; }
}
```

Le serveur défini dans l'exemple précédent ne permet pas de traiter plus d'une requête à la fois.

Pour pouvoir accepter plusieurs requêtes simultanément il est nécessaire d'utiliser un **Thread** différent pour gérer la connexion de chaque nouvelle requête.

```
import java.io.*
import java.net.*

public class Serveur {
           private ServerSocket sock ;
           private Socket client ;

           Serveur() { try {
                      sock = new ServerSocket(5000) ;
                      while(true) {
                                 client = sock.accept() ;
                                 Requete req = new Requete(client) ;
                                 req.start() ;
                      }
           } catch (Exception e) { System.out.println(e) ; }
           }

           public static void main(String argv[]) { new Serveur( ) ; }
}
```

```
import java.io.*
import java.net.*

public class Requete extends Thread {
        private Socket sock ;

        Requete(Socket sock) { this.sock = sock ; }

        public void run() {
                BufferedReader in =
                        new BufferedReader(new InputStreamReader(sock.getInputStream())) ;
                PrintWriter out =
                        new PrintWriter( new OutputStreamWriter(sock.getOutputStream()),true);

                String input = in.readLine();
                if (input != null) {
                        System.out.println(input) ;
                        out.println("Bonjour Client de la part de " + InetAddress.getLocalHost() ) ;
                        }
        }
}
```

# 14. Les applets

## 14.1 Applets et Web

Les applets sont une particularité de Java. Ce sont de petites applications que l'on peut charger à travers le réseau dans un navigateur Web qui intègre une machine virtuelle Java capable d'exécuter le byte code ainsi chargé…

Une traduction possible de applet en français est appliquette ou applicule.

De manière concrète les applets sont chargées et appelées à travers un fichier HTML Dans le code source de ce fichier HTML, un **tag APPLET** permet de spécifier la classe de base de cette applet et les paramètres nécessaires à son lancement. La machine virtuelle se charge alors de récupérer les classes utiles à son exécution. Seules quelques classes sont réellement chargées, les cla sses du JDK sont déjà présentent dans le navigateur…

Exemple de tag APPLET :

```
…
<APPLET
        CODE = "MonApplet.class"
        NAME = "MonApplet"
        WIDTH = 200
        HEIGHT = 200
        CODEBASE = "http://www.monserveur.org/java/"
        ALT = "Desole, Votre navigateur ne supporte pas Java"
        >
        <PARAM  NAME = "Param1" VALUE = "toto">
        <PARAM  NAME = "Param2" VALUE = "tata">
        <PARAM  NAME = "Param3" VALUE = "titi">
</APPLET>
```

CODE : le nom de la classe
NAME : nom indicatif de l'applet
WIDTH : largeur de la zone d'affichage de l'applet dans la page
HEIGHT : hauteur de la zone d'affichage de l'applet dans la page
CODEBASE : répertoire contenant l'arborescence de classes sur le serveur
ALT : A afficher si l'applet ne peut être exécuté par le navigateur (il ne supporte pas Java)
PARAM : permet de définir un paramètre à passer à l'applet (nom et valeur)

## 14.2 La classe java.applet.Applet

Toute classe définissant une applet est une classe fille de la classe **Applet**.

La classe Applet hérite de la classe **java.awt.Panel** et par conséquent de **java.awt.Container** : il est donc possible d'ajouter dans une Applet n'importe quel composant de l'AWT (le gestionnaire d'apparence par défaut d'une Applet est le **FlowLayout**).

Il n'y a pas de fonction ***main()*** dans une Apple, mais rien n'empêche de placer une méthode ***main()*** dans une classe héritant de Applet, afin de pouvoir démarrer l'applet comme une application autonome. Cependant il est important de comprendre que cette méthode ne sera pas utilisée par un navigateur pour démarrer l'applet.

A la place de la fonction ***main()***, un navigateur utilisera quatre méthodes. Il s'agit de :

**void init() ;**
**void start() ;**
**void stop() ;**
**void destroy() ;**

Lorsqu'une applet est chargée, voici comment le navigateur interagit avec elle :

- Le ***constructeur*** de la classe qui hérite de Applet est appelé en premier
- La méthode ***init()*** est appelée immédiatement après la création de l'applet. C'est en principe là que l'on charge et crée tout ce dont on a besoin pour l'exécution de l'applet.
- La méthode ***start()*** est appelée après ***init()*** et chaque fois que l'applet est interrompue et réactivée (par exemple lorsque l'on quitte la page HTML contenant l'applet et que l'on revient dessus).
- La méthode ***stop()*** est appelée chaque fois que l'on interrompt l'applet (***start()*** est ensuite appelée si on la réactive) .
- La méthode ***destroy()*** est appelée lorsque l'on quitte le navigateur.
- La méthode ***paint()*** est appelée chaque fois que l'on redessine le navigateur.

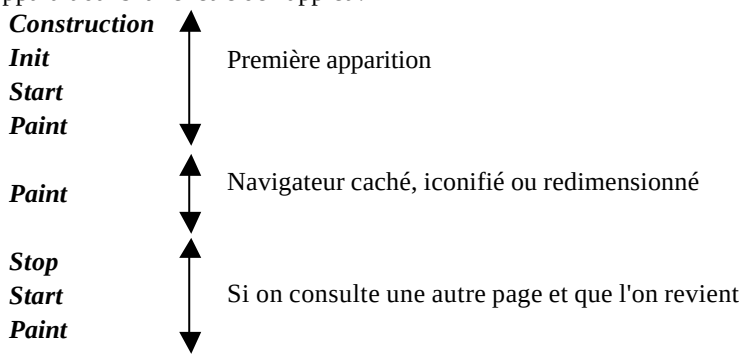
```
import java.applet.* ;
import java.awt.* ;
import java.util.* ;

public class MontreTrace extends Applet {
        private Vector trace = new Vector() ;

        public MontreTrace() { trace.addElement( "Construction" ) ; }
        public void init() { trace.addElement("Init") ; }
        public void start() { trace.addElement("Start") ; }
        public void stop() { trace.addElement("Stop") ; }
        public void destroy() { trace.addElement("Destroy") ; }

        public void paint(Graphics g) {
                int y = 20 ;
                trace.addElement("Paint") ;
                for ( Enumeration e = trace.elements() ; e.hasMoreElements() ; y += 10)
                        g.drawString( (String)e.nextElement(), 10, y) ;
        }
}
```

Ce qui apparaît dans la fenêtre de l'applet :




## 14.3 Méthodes utiles de la classe Applet

Voici quelques méthodes utiles :

**String getAppletInfo() ;** // Informations sur l'applet
**URL getCodeBase() ;** // Permet de récupérer l'URL du répertoire où a été chargée l'applet
**URL getDocumentBase() ;** // Permet de récupérer l'URL du document HTML contenant l'applet

Nous avons vu qu'il est possible de passer des paramètres à une applet. Il est p ossible de récupérer ces paramètres grâce à la méthode :

**String getParameter(String NomDuParametre) ;** // Retourne la valeur du paramètre

Il est préférable d'utiliser cette méthode dans la méthode ***init()*** plutôt que dans le constructeur.

```
<APPLET
        CODE = "testparam.class"
        WIDTH = 200
        HEIGHT = 200 >
                <PARAM NAME = "Largeur" VALUE = "10">
</APPLET>
```

```
public void init() {
        …
        String largeurPar = getParameter("Largeur") ;
        int largeur ;

        if (largeurPar != null) largeur = Integer.parseInt(largeurPar) ;
        else largeur = 20 ;
        …
}
```

## 14.4 Applets et sécurité

Pour des raisons de sécurité sur Internet, une Applet chargée à partir d'un serveur distant sur le réseau subit de nombreuses restrictions lors de son exécution (ce n'est pas le cas si elle est chargée à partir du disque local). Ainsi, par défaut, une applet ne peut pas :

- Lire ou écrire sur des fichiers locaux du poste client
- Communiquer d'une manière ou d'une autre avec un autre serveur que celui dont elle est issue.
- Exécuter un programme sur le système client (ou appeler une DLL, ce qui revient au même)

En fait, elle ne peut accéder qu'aux ressources du serveur dont elle est issue…

Depuis le JDK 1.1, il est possible de signer une applet et ainsi de lui accorder des droits supplémentaires. Cependant cette facilité est implémentée de manière incompatible dans les principaux navigateurs du marché, avec pour résultat de rendre inutilisable ce procédé. Cela devrait être résolu dans la version 1.2 du JDK

## 14.5 Applets et threads

Pour montrer la nécessité d'utiliser des threads dans les applets, étudions une applet implémentant une horloge digitale.

Dans sa méthode ***start()*** cette Applet construit toutes les secondes un objet de la classe **Date** (qui par défaut est construit avec la date et l'heure de l'instant) et invoque la méthode ***repaint()*** pour mettre à jour son dessin (appel indirect de ***paint()***).
De son coté la méthode ***paint()*** permet de mettre à jour l'affichage de cette date.

```
public class Horloge extends Applet {
        private Font font = new Font("TimesRoman", Font.BOLD, 24) ;
        private Date  maintenant ;

        public void start() {
                while(true) {
                        maintenant = new Date() ;
                        repaint() ;
                        Thread.sleep(1000) ;
                }
        }

        public void paint(Graphics g) {
                g.setFont(font) ;
                if( maintenant != null ) g.drawString(maintenant.toString(), 10, 50) ;
        }
}
```

Tout semble parfait, **mais… cela ne marche pas !**

En effet, lorsqu'un navigateur démarre une Applet, il la lance dans son propre Thread : si l'Applet ne rend pas la main au navigateur, celui-ci sera bloqué. Dans notre cas, la méthode start() contient une boucle infinie qui va donc monopoliser toutes les ressources du système.

En fait, en fonction du navigateur, les résultats risquent d'être différents (soit l'applet ne fonctionne pas, car le navigateur reprend la main de force, soit l'applet s'exécute mais le navigateur ne répond plus…)

➔ **Il est impératif de rendre la main au plus tôt en créant un nouveau thread chargé d'effectuer le travail**

Voici donc une applet qui marche :

```
public class Horloge extends Applet implements Runnable {
        private Font font = new Font("TimesRoman", Font.BOLD, 24) ;
        private Date  maintenant ;
        private Thread horloge ;

        public void run() {
                while(true) {
                        maintenant = new Date() ;
                        repaint() ;
                        try { Thread.sleep(1000) ; } catch (InterruptedException e) { }
                }
        }
        public void paint(Graphics g) {
                g.setFont(font) ;
                if( maintenant != null ) g.drawString(maintenant.toString(), 10, 50) ;
        }
        public void start() {
                horloge = new Thread(this) ;
                horloge.start() ;
        }
        public void stop() ;
                horloge.stop() ;
                horloge = null ;
        }
}
```

## 14.6 L'interface AppletContext et la méthode getAppletContext()

### 14.6.1 Présentation

Toute applet peut disposer d'un objet de type **AppletContext** qui représente le contexte dans lequel s'exécute cette applet. L'objet **AppletContext** permet en fait d'interagir avec le navigateur.

Pour récupérer cet objet, il suffit d'utiliser la méthode de la classe **Applet** :

**AppletContext getAppletContext() ;**

Les méthodes les plus intéressantes de l'**AppletContext** sont :

**void showStatus(String mess) ;** // permet d'afficher un message dans la barre de status du navigateur

**void showDocument(URL u) ;** // permet d'afficher le document désigné par u dans la fenêtre courante du navigateur

**void showDocument(URL u, String fenêtre) ;** // Idem, mais le document est affiché dans la fenêtre désignée par le deuxième argument (par exemple si l'on est dans une page multi-fenêtre) ou dans une nouvelle fenêtre si ce nom n'existe pas déjà. Il existe 4 noms spéciaux pour désigner des fenêtres:

- **"_self" :** *La fenêtre courante*
- **"_parent" :** *La fenêtre parente*
- **"_top" :** *la fenêtre principale (page multi-fenêtre)*
- **"_blank" :** *Nouvelle fenêtre sans titre*

**Image getImage(URL u) ;** // permet de charger une image dans un objet Image (supporte les formats GIF et JPEG)

**AudioClip getAudioClip(URL u) ;** // permet de charger une séquence audio

**Applet getApplet(String nom) ;** // permet de récupérer une référence sur l'applet dont le nom est passé en argument. Cette applet doit être chargé dans la même page que l'applet courante. Cette méthode permet de faire de la communication inter-applets.

### 14.6.2 Exemple : Interactions avec le navigateur

Applet permettant d'afficher une série de 5 images (picture0.gif à picture4.gif) en tant que document.

```
import java.applet.* ;
import java.net.* ;
public class ImagesNav {
        private Thread navigateur ;
        public void run() {
                AppletContext contexte = getAppletContext() ;
                for (int i=0 ; i<5 ; i++ ) {
                        try {
                                String nimage = "picture" + i + ".gif" ;
                                URL uimage = new URL(getCodeBase.toString() + nimage) ;
                                contexte.showDocument(uimage) ;
                                contexte.showStatus(nimage) ;
                                Thread.sleep(1000) ;
                        }
                        catch (MalFormedURLException e) { }
                        catch (InterruptedException e) { }
                }
        }
        public void start() { (navigateur = new Thread(this)).start() ;    }
        public void stop() { navigateur.stop() ; horloge = null ; }
}
```

### 14.6.3 Exemple : Communication inter-applets

Nous présentons ici une applet affichant un bouton pour arrêter une applet de la classe Horloge que nous avons définie précédemment. Ces deux applets doivent être chargées par le même document HTML.

```
import java.applet.* ;
import java.awt.* ;
import java.awt.event.* ;

public class MonBoutonStop extends applet implements ActionListener {
        private Button bouton ;
        private String nomAppCible ;
        public void init() {
                nomAppCible = getParameter("NomApplicationCible");
                bouton = new Button( getParameter("NomBouton"));
                add(bouton) ;
                bouton.addActionListener( this ) ;
                }
        public void actionPerformed(ActionEvent e) {
                Applet app = getAppletContext().getApplet(nomAppCible) ;
                app.stop() ;
        }
}
```

Extrait du document HTML nécessaire pour mettre en œvre cette applet :

```
<HTML>
…
<APPLET          CODE="MonBoutonStop.class"
                 HEIGHT=100
                 WIDTH=100
                 ALIGN="MIDDLE">
        <PARAM  NAME="NomAPplicationCible" VALUE="Clock">
        <PARAM  NAME="NomBouton" VALUE="Stop Horloge">
</APPLET>
<APPLET          CODE="Horloge.class"
                 NAME="Clock"
                 HEIGHT=100
                 WIDTH=300
                 ALIGN="MIDDLE">
</APPLET>
```

### 14.6.4 Exemple : Chargement d'images

Affichage d'une image dans une Applet :

```
public class ChargeImage extends Applet {
        private Image image ;
        public void init() { image = getImage(getCodeBase(), "image.gif") ; }
        public void paint(Graphics g) {
                g.drawImage(image, 10, 10, this) ;
        }
}
```