



Université Sidi Mohamed Ben Abdellah
Faculté des Sciences Dhar El-Mehraz Fès
Département d'Informatique

Le Langage Java

Par Nouredine Chenfour
2002-2015



Chapitre 1. Environnement de développement Java

1.1 Introduction

Java est un langage de programmation à objets développé par Sun Microsystems. Sa première version a été publiée en 1995.

Le langage Java est un langage qui dérive du C/C++ tout en imposant un seul mode de programmation : par les objets. Java dispose d'un mécanisme de gestion de mémoire très puissant. En effet, tous les objets gérés par un programme Java sont créés dynamiquement. De plus, périodiquement ou à chaque fois qu'on a besoin de la mémoire un programme appelé Garbage-Collector (ou ramasse-miettes) est exécuté automatiquement pour récupérer la mémoire qui n'est plus utilisée. Java est aussi un langage multitâche. Il permet la création et la gestion des processus légers (les Threads).

Java dispose d'une gigantesque bibliothèque de classes spécialisées dans différents thèmes et différents domaines (de la gestion des chaînes de caractères jusqu'au cryptage et sécurité des réseaux en passant par des bibliothèques graphiques, bibliothèque mathématique, accès aux bases de données JDBC, synchronisation des processus par le principe des moniteurs et le mécanisme des signaux, etc...)

1.2 Environnement de développement

1.2.1 Langage Java

Il existe deux modes de programmation en Java : le mode Applications et le mode Applets :

Le mode applications est le mode ordinaire des différents langages de programmation.

Le mode applets permet de créer des applications pouvant être incorporées dans des pages Web. Les applets sont en fait des applications particulières dont le déroulement est prévu sur une page Web (le conteneur est une page Web et non une fenêtre ordinaire). Le programme applet est ainsi interprété et exécuté par le navigateur qui doit tout d'abord récupérer le programme de la même manière qu'il récupère par exemple un fichier image avant de l'afficher. Une applet peut aussi être exécutée avec un programme spécial (délivré par sun) appelé AppletViewer. Il permet ainsi de tester le fonctionnement d'une applet avant sa diffusion sur Internet.

1.2.2 Portabilité de Java

Le mode de compilation Java constitue la première particularité de ce langage. En effet, un programme Java (extension .java) est traduit par le compilateur Java (programme **javac.exe**) non pas en langage machine (programme exécutable) mais en un pseudo code intermédiaire appelé **Bytecode** indépendant de toute machine ou plate-forme. Le bytecode ainsi généré (sous forme de fichier d'extension **.class**) peut ensuite être interprété sur n'importe quelle machine ou plate-forme. Il suffit que celle-ci dispose d'un programme de traduction adéquat. Celui-ci constitue en fait ce qu'on appelle « **Machine Virtuelle Java** » (ou **JVM : Java Virtual Machine**). L'un des composants de la machine virtuelle Java est le programme **java.exe** permettant d'exécuter directement un programme **.class** (en faisant appel à d'autres bibliothèques nécessaires à l'exécution).

La machine virtuelle Java, quant à elle, dépend alors de la plate-forme ou la machine sur laquelle elle tourne. En fait, Sun publie sur son site différentes machines virtuelles destinées à différentes plates-formes de développement.

Le schéma de développement et d'exécution d'un programme Java est alors le suivant :

On écrit un programme java sauvegardé dans un fichier d'extension .java (exemple Test01.java : supposé contenir une classe nommée Test01)

On compile ensuite le programme pour générer le Byte Code associé :

```
C:\ ... > javac Test01.java
```

Un fichier Test01.class est alors créé.

Exécution du programme résultant :

```
C:\ ... > java Test01
```

Il est à remarquer qu'un même programme Java est généralement organisé sous forme de plusieurs classes successives (dont l'une fait appel à des instances de l'autre). Le compilateur Java génère alors un fichier **.class** associé à chacune des classes du programme est non au programme lui même. Si ainsi le programme Test01.java contient deux classes nommées C1 et C2 (et pas de classe Test01), le compilateur génère deux fichiers de classes : C1.class et C2.class (et pas de fichier Test01.class). Ce qu'il faut alors exécuter par la suite est la classe principale (c à d : celle contenant une méthode **main**) par exemple C1 :

```
C:\ ... > java C1
```

1.2.3 Le JDK (Java Développement Kit)

1.2.3.1 Définition

Le JDK est l'ensemble des programmes nécessaires pour le développement d'applications Java. Il regroupe ainsi les programmes javac.exe, java.exe,

appletviewer.exe pour exécuter les applets, ainsi que d'autres classes et utilitaires de développements.

1.2.3.2 Evolution

Depuis 1995 jusqu'à aujourd'hui le JDK n'a cessé d'évoluer et d'être étendu de version en version : 1.0, 1.1, 1.2, 1.3, 1.4, ...

1. La version 1.0 (lancée en 1995 et constituée de quelques 212 classes et interfaces) a subi quelques évolutions jusqu'à sa dernière version : 1.0.2. Cette notation 1.s.b est interprétée avec le chiffre b indiquant des corrections de bugs ou des petites modifications alors que le chiffre au milieu s indique des modifications de spécification ou des améliorations de taille (ajout de classes et interfaces).
2. Le JDK 1.1 (504 classes et interfaces) a subi lui même beaucoup de modifications jusqu'à sa dernière version 1.1.8.
3. Le JDK 1.2 (1692 classes et interfaces) annoncé en décembre 1998 a évolué avec sa dernière version 1.2.2. Nom de code Playground
4. Le JDK 1.3 (76 packages et 1839 classes et interfaces). Nom de code Kestrel
5. Le JDK 1.4 (Nom de code Merlin, avec 139 packages et 2723 classes et interfaces)
6. Le JDK 1.5 (ou JDK 5, nom de code Tiger, avec 166 packages et 3279 classes et interfaces)
7. Java 6.0 (Nom de code Mustang), Octobre 2006 (11 Décembre): 3777 classes et interfaces)
8. En fin, le dernier né du groupe (Fin 2010) : Java 7.0 (Nom de code Dolphin) sous licence GPL.

1.2.3.3 Java™ 2

Nous remarquons toutefois que Sun a voulu changer la nomination du JDK en la remplaçant à partir de sa version 1.2 par SDK (Software Développement Kit) ou plus précisément J2SDK pour « Java™ 2 SDK ». Avec Java 2 désignant le JDK 1.x ($x \geq 2$). En fait Sun distingue entre deux concepts :

La spécification proposée ou exigée par Sun indépendamment des détails d'implémentation du langage ou la plate-forme de développement. Celle-ci est notée : J2SE (Java™ 2 platform Standard Edition).

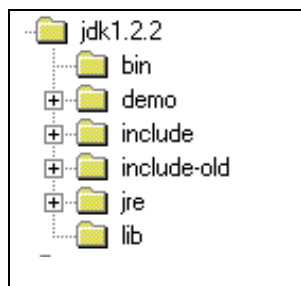
Le software ou le kit qui implémente la spécification : J2SDK.

Pour terminer la remarque, on résume que JDK, SDK, Java 2, J2SDK sont tous des termes qui désignent la même chose : le kit de développement Java.

1.2.3.4 Installation sur le disque

Généralement le JDK (ou SDK) est délivré (ou téléchargé gratuitement à partir du site de Sun) sous forme d'un programme dont l'exécution conduit à l'installation du kit.

Si l'on prend l'exemple du JDK 1.2.2, son installation par défaut créera l'arborescence suivante (de quelques 25 Mo) :



Tous les programmes dont on a besoin pour développer (javac, java, ...) se trouveront alors dans le répertoire : `jdk1.2.2\bin` qu'il est possible d'ajouter au path pour être accessible depuis n'importe quel répertoire du disque.

1.2.4 Le JRE (Java Runtime Environment)

Le JRE est un ensemble d'outils de classes et de bibliothèques (.dll) nécessaires pour l'exécution d'un programme Java. Ainsi le déploiement ou la diffusion d'une application développée en Java nécessite que la machine sur laquelle l'application est installée dispose d'une JVM. Le JRE constitue en quelque sorte cette machine virtuelle ainsi que tous les outils dont a besoin toute application Java pour pouvoir fonctionner. Le JRE est lui aussi accessible gratuitement dans le site de Sun. Lorsqu'il est installé sur le disque, il se présente sous forme d'un répertoire JRE (de quelques 20 Mo pour le JRE 1.2.2 associé au JDK 1.2.2).

1.2.5 Les Packages

Nous avons déjà mentionné que le JDK dispose de plusieurs classes prêtes à être utilisées. Ces classes sont en effet délivrées avec le JDK ou le JRE. On y trouve même la version source de ces classes dans un fichier archivé (src.jar).

Les classes accessibles sont généralement organisées en des arborescences. Ainsi par exemple la classe String se trouve dans le sous répertoire java\lang. Ce dernier est appelé paquetage ou Package. Ainsi lang est un package qui se trouve dans le package java. D'autres sous packages du package java peuvent être signalés tels que : java.util, java.awt, etc.. On remarque l'utilisation du symbole point « . » comme séparateur de chemin pour assurer la portabilité entre systèmes : le point sera remplacé par le symbole anti-slash « \ » sous Windows et « / » sous Unix.

Dans un programme java, pour utiliser une classe d'un certain package par exemple la classe Vector du package java.util on démarre le programme avec la ligne suivante :

```
import java.util.* ;
```

Ce qui signifie que toutes les classes du package sont accessibles. Ou encore :

```
import java.util.Vector ;
```

pour n'accéder qu'à la classe **Vector**.

Remarques :

Le Package `java.lang` est un package par défaut. Ainsi il est possible d'utiliser toutes les classes de ce package sans avoir besoin de le mentionner par une commande `import`.

Afin d'optimiser l'espace disque occupé par les packages, Sun a mis en place un mécanisme d'archivage ou compression de tous ces packages (avec leur classes) sous forme d'un fichier d'extension `.jar` (`rt.jar`). Les classes et les packages restent comme même accessibles comme s'ils n'étaient pas archivés. On remarque que le JDK comporte un programme `jar.exe` (dans le sous-répertoire `bin`) avec lequel le programmeur peut fabriquer ces propres fichiers `.jar`.

1.3 Eléments de base du langage Java

1.3.1 Conventions de nommage

Il n'est pas obligatoire mais il est recommandé que le programmeur respecte un certain ensemble de règles lors de l'écriture d'un programme. Ces mêmes règles sont d'ailleurs respectées par les développeurs de Sun, ce qui facilite la lecture des programmes et permet de deviner l'écriture d'un nom de classe ou de méthode quelconque. Ces règles peuvent être résumées de la manière suivante :

Les noms de packages sont entièrement en minuscule. Exemples :

- **`java.awt`**
- **`javax.swing`**
- **`javax.swing.filechooser`**
- **`etc...`**

Un nom de classe est une séquence de mots dont le premier caractère de chaque mot est en majuscule et les autres en minuscule.

Exemples :

- **String**
- **StringBuffer**
- **ComboBoxEditor**

Un nom de méthode est une séquence de mots dont le premier caractère de chaque mot est en majuscule et les autres en minuscule sauf le premier mot qui est entièrement en minuscule. Exemples :

- **append**
- **toString**
- **deleteCharAt**

Une propriété est en principe un membre privé donc non accessible directement et par suite on peut lui choisir un nom librement. Sinon on lui applique la même règle que celle d'une méthode.

Une constante (**final**) est une séquence de mots majuscules séparés par un blanc souligné « **_** »

- **PI**
- **MIN_VALUE**
- **MAX_VALUE**

Les primitives (types de base ou type primitif) et les mots clés sont en minuscule :

byte, int, ...

while, for, if

this, super, try, catch, length, ...

class, extends, implements, null, ...

1.3.2 Différences lexicales et syntaxique avec le langage C++

Le langage Java a éliminé plusieurs structures du langage C/C++ dont on peut citer :

- le type enum (avant la version 5)
- le mot clé const remplacé par **final**
- le mot clé goto
- la notion de variables globales
- les unions
- les instructions #include et #define
- les opérateurs : *, &, ->
- la désallocation
- liste d'arguments variables (avant la version 5)

1.3.3 Structure générale d'un programme Java

Le squelette général d'un programme Java se présente comme suit (chaque classe dans un fichier séparé et portant le même nom que celui de la classe) :

```
public class NomDeClasse1 {  
    // Propriétés de la classe  
    // Méthodes de la classe  
}
```

```
public class NomDeClasse2 {  
    // Propriétés de la classe  
    // Méthodes de la classe  
}
```

...

```
public class NomDeClassePrincipale {  
    // Propriétés de la classe  
    // Méthodes de la classe  
    public static void main(String args[]) {  
        // Code du programme principal  
    }  
}
```

Il est à remarquer que le compilateur Java génère un fichier de classe (.class) pour chacune des classes qui constituent le programme. Il est donc d'usage d'écrire chaque classe séparément (dans un fichier séparé portant le même nom que celui de la classe et avec respect de la casse). Le compilateur n'aura pas de problème pour retrouver la classe compilée (si elle est référencée à partir d'une autre classe qui se trouve dans un autre fichier) tant qu'on travaille dans le même répertoire (ou plus exactement dans le même **package**). Dans le cas contraire (différents packages), on utilise l'instruction **import** pour préciser le chemin des classes accessibles.

1.3.4 Modificateurs et visibilité

Nous remarquons que la méthode **main()** qui constitue le point d'entrée de toute application est précédée par 3 mots réservés **public static** et **void**. Le mot clé **void** désigne le type de retour, c'est donc une procédure. **public** et **static** sont des modificateurs. Il existe 5 types de modificateurs très utilisés qui peuvent être associées à une données ou une fonction membre : modificateurs de synchronisation, de visibilité, de permanence, de constance et d'abstraction.

1- Synchronisation	2- Visibilité	3- Permanence	4- Constance	Type	Nom
synchronized (seulement avec les méthodes)	public - protected private	static	final	void int ...	
5- Abstraction					
abstract (seulement les méthodes qui ne sont pas synchronized , static , final ou private)					

Exemples :

- **synchronized public static final** void p1() {...}
- **public abstract** void p2() {...}
- **public static** int f1(){...}

Modificateur synchronized :

Permet de mettre en place une méthode ou un bloc de programme verrouillé par l'intermédiaire du mécanisme de moniteur. Une méthode **synchronized** est une méthode dont l'accès sur un même objet est réalisé en exclusion mutuelle.

Modificateurs private, public et protected :

Une donnée ou méthode **private** est inaccessible depuis l'extérieur de la classe où elle est définie même dans une classe dérivée.

Une donnée ou méthode **public** est accessible depuis l'extérieur de la classe où elle est définie

Une donnée ou méthode **protected** est protégée de tout accès externe comme **private** sauf à partir des classes dérivées. Une méthode ou donnée **protected** est donc accessible dans une classe fille.

Remarque :

Lorsqu'on **redéfinit** une méthode dans une classe fille, il est obligatoire de conserver son modificateur de visibilité ou d'utiliser un privilège d'accès plus fort. Càd :

une méthode **public** reste **public**,

une méthode **protected** reste **protected** ou devient **public**,

et une méthode **private** reste **protected** ou devient **public** ou **protected**.

Modificateur static :

Le modificateur **static** permet de définir une donnée ou une méthode commune à toutes les instances de la classe. Ce sont des entités qui existent en l'absence de tout objet de la classe. Ainsi, une donnée ou méthode **static** est associée à sa classe entière, et non à une instance particulière de la classe. Le mot clé **static** peut aussi être appliqué à toute la classe (**static class ...**, mais uniquement pour les classes internes) dans tel cas tous les membres de la classe seront **static**.

Les membres static peuvent donc être appelés directement par l'intermédiaire du nom de la classe. On n'a pas besoin de créer une instance de la classe pour accéder à une méthode ou donnée static se trouvant dans cette classe. L'appel est réalisé simplement à l'aide du sélecteur « . » (le C++, utilise pour ce cas le sélecteur « :: »).

Les données static sont appelées **variable de classe** et les méthodes static sont également appelées **méthodes de classe**. Un exemple concret de méthodes static est la méthode **main()** qui doit être static afin de rester indépendante des autres objets que le programme peut générer à partir de sa classe principale.

*** Remarque :**

Puisque les données non-static d'une classe n'ont d'existence que lorsqu'on a instancier un objet de la classe et inversement une méthode static existe sans avoir besoin de créer d'instance de classe, **une méthode static ne peut donc pas accéder à une donnée non-static ou une autre méthode non-static :**

Exemple :

```
class Class1 {  
    static int x = 20;  
    int y = 30;  
    static void p1() {  
        x = x + 1 ; // correcte  
        y = y * 2 ; // incorrecte  
        p2(); // incorrecte  
    }  
  
    void p2() { // méthode non-static  
        x ++ ; // correcte  
        y ++ ; // correcte  
        p1() ; // correcte  
    }  
}
```

```
    }  
}
```

Modificateur final :

Les données **final** sont des constantes. Ces données sont généralement définies en plus public et static afin d'être accessibles depuis l'extérieur de la classe et directement par l'intermédiaire du nom de celle-ci sans avoir besoin de créer une instance de la classe.

Exemple :

```
class Constantes {  
    public static final int CONST1 = 20;  
    public static final double PI = 3.14;  
}
```

Lorsqu'une méthode porte le modificateur **final**, elle ne peut pas être redéfinie dans une classe fille.

Modificateur abstract :

Ce modificateur permet de définir une méthode abstraite. C'est une méthode dont le corps n'est pas défini. Une classe qui comporte une méthode abstraite doit, elle aussi, être définie abstraite. Il s'agit d'une classe de spécification qui ne peut être directement instanciée. Elle nécessite d'être redéfinie dans une classe fille qui aura donc l'objectif d'implémenter les spécifications de la classe abstraite mère.

La définition de méthodes abstraites est réalisée de la manière suivante :

```
abstract class ClassAbstraite {  
    ...  
    abstract typeretour methodeAbstraite(parametres) ;  
}
```


Remarque :

Vue la notion d'abstraction, une méthode abstraite ne peut pas être :

- **synchronized** : car elle n'est pas encore définie
- **static** : pour la même raison (elle n'a pas encore d'existence, or une méthode static existe dès la création de la classe).
- **final** : car cela va empêcher sa redéfinition, or une méthode abstract n'a d'existence que lorsqu'elle est définie
- **private** : la redéfinition d'une méthode private signifie en fait la définition d'une nouvelle méthode qui porte le même nom (et non pas le véritable sens de la redéfinition). De ce fait on ne peut pas donner d'existence à une méthode abstraite déclarée private. Ce qui interdit la combinaison **private-abstract**.

1.3.5 La méthode main

La remarque que nous avons fait à propos des méthodes static reste valable pour le **main** qui est une méthode **static**. Cela impose une grande restriction, car la méthode main ne peut accéder qu'à des données et des méthodes static. Pour résoudre ce problème, il suffit de créer un objet de la classe principale dans la méthode main. Ce qui implique l'exécution du constructeur. Ce dernier n'est pas une méthode static on peut alors y mettre tous les accès nécessaires aux différents membres non-static. Le squelette d'un programme Java est souvent alors formé de la manière suivante :

```

class NomDeClasse {
    // données membres
    NomDeClasse () { // Constructeur
        ...
    }
    ...
    public static void main(String args[]) {
        new NomDeClasse();
    }
}

```

Exemple:

```

class Class1 {
    static int x = 20;
    int y = 30;
    Class1() {
        x = x + 1 ; // correcte
        y = y * 2 ; // correcte
        p2() ; // correcte
    }
    void p2() {
        System.out.println("x = " + x + "    y = " + y) ;
    }

    public static void main(String args[]) {
        new Class1();
        y = y * 2 ; // incorrecte
        p2(); // incorrecte
    }
}

```

1.3.6 Unités lexicales et syntaxiques de base du langage

Les unités de base du langage Java sont héritées du langage C. Ainsi le langage Java utilise sans modification les entités et structures suivantes du langage C :

- **Affectation** : `=`
- **Opérateurs arithmétiques** : `+, -, *, /, %, ++, --, +=, -=, *=, /=, %=, ...`
- **Opérateurs de relation** : `<, <=, >, >=, !=, ==`
- **Opérateurs sur le bit** : `&, |, ~, ^, <<, >>, >>>`
- **Instructions de choix** :
 - `if (condition) traitement1 ; else traitement2 ;`
 - `switch (...) { case ... default }`
- **Opérateurs logiques** : `&&, ||, !`
- **Les boucles** :
 - `for (...) {...}`
 - `while (condition) {...}`
 - `do { ... } while (condition)`
- **Transtypage** :
 - `variable = (type) Expression ;`
- **Commentaires** :
 - `//...`
 - `/* ...*/`

Chapitre 2. Programmation Orientée objet en Java

2.1 Principe de base de la POO

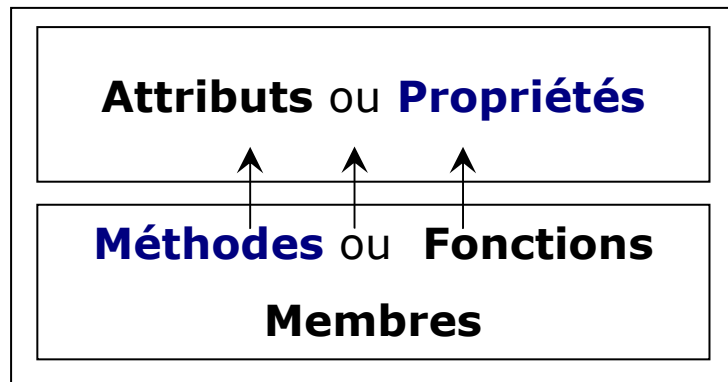
Une structuration classique d'un programme consiste en une structuration à deux niveaux : les données d'une part et le code d'une autre part. Ainsi les données qui décrivent ou caractérisent une même entité sont regroupées ensemble dans une même structure de donnée : un enregistrement ou un tableau. De la même manière des instructions réalisant ensemble une tâche bien définie et complète sont regroupées dans une même procédure ou fonction.

La Programmation Orientée Objets (**POO**) consiste en une structuration de plus haut niveau. Il s'agit de regrouper ensemble les données et toutes les procédures qui permettent la gestion de ces données. On obtient alors des entités comportant à la fois un ensemble de données et une liste de procédures et de fonctions pour manipuler ces données. La structure ainsi obtenue est appelée : **Objet**.

Un objet est alors une généralisation de la notion d'enregistrement. Il est composé de deux parties :

- Une partie statique (fixe) composée de la liste des données de l'objet. On les appelle : **Attributs** ou **Propriétés**, ou encore : **Données Membres**.
- Une partie dynamique qui décrit le comportement ou les fonctionnalités de l'objet. Elle est constituée de l'ensemble des procédures et des fonctions qui permettent à l'utilisateur de configurer et de manipuler l'objet. Ainsi les données ne sont généralement pas accessibles directement mais à travers

les procédures et les fonctions de l'objet. Celles-ci sont appelées : **Méthodes** ou **Fonctions Membres**.



2.2 Encapsulation

Un objet tel que défini plus haut est une variable caractérisée par des propriétés et des méthodes. La définition de ces propriétés et de ces méthodes devra être réalisée dans une structure de données appelée **classe**.

Une **classe** est donc le support de l'encapsulation : c'est un ensemble de données et de fonctions regroupées dans une même entité. Les données seront généralement appelées des propriétés, les fonctions qui opèrent sur ces propriétés sont appelées des méthodes.

Créer un objet depuis une classe est une opération qui s'appelle **l'instanciation**. L'objet ainsi créé pourra être appelé aussi : **instance**. Entre classe et objet il y a, en quelque sorte, le même rapport qu'entre type de données et variable.

Pour utiliser une classe il faut en déclarer une instance de cette classe (appelée aussi objet).

2.3 Syntaxe de déclaration d'une classe

```
class NomDeClasse {  
    Propriétés de la classe  
}
```

Méthodes de la classe

}

L'ordre des méthodes dans une classe n'a pas d'importance. Si dans une classe, on rencontre d'abord la méthode `m1()` puis la méthode `m2()`, `m2()` peut être appelée sans problème dans `m1()`.

2.4 Instanciation

Il est nécessaire de déclarer un objet avant son instanciation. La déclaration est réalisée de la manière suivante :

NomDeClasse nomDeVariable ;

L'instanciation est réalisée ensuite à l'aide de l'opérateur `new` qui se charge de créer une instance (ou un objet) de la classe et de l'associer à la variable

nomDeVariable = new NomDeClasse();

Il est possible de réunir les 2 instructions en une seule :

NomDeClasse nomDeVariable = new NomDeClasse();

Si `objet1` désigne un objet de type `NomDeClasse`, et si `objet2` est un nouvel objet créé à l'aide de l'instruction `objet2 = objet1`, il est alors à noter que cette affectation ne définit pas un nouvel objet mais `objet1` et `objet2` désignent tous les deux le même objet.

L'opérateur `new` est un opérateur de haute priorité qui permet d'instancier des objets et d'appeler une méthode particulière de cet objet : le **constructeur**. Le constructeur est une méthode de classe (sans valeur de retour) qui port le même nom que celui de la classe et dans laquelle sont réalisées toutes les initialisation de l'objet en cours.

Le constructeur peut avoir des paramètres et dans tel cas l'instanciation est réalisée avec des arguments d'appel :

NomDeClasse nomDeVariable = new NomDeClasse(a1, a2, ...);

Exemple :

String S = new String("abc");

Pour le cas des chaînes de caractères, l'écriture précédente et équivalente à l'instruction suivante :

String S = "abc";

Les constantes chaînes sont des objets String

2.5 Durée de vie d'un objet

Les objets en java sont tous dynamiques. La durée de vie d'un objet passe par trois étapes :

- 📄 la déclaration de l'objet et son instanciation à l'aide de l'opérateur new
- 📄 l'utilisation de l'objet en appelant ces méthodes
- 📄 la suppression de l'objet : elle est réalisée automatiquement à l'aide du garbage collector qui libère l'espace mémoire associé à l'objet ne référence plus cet espace mémoire (par exemple affectation de **null** à l'objet) et lorsque cet espace mémoire n'est plus référencé par aucune autre variable).

Il n'existe pas d'instruction delete comme en C++.

2.6 Affectation d'objets

Exemple :

NomDeClasse objet1 = new NomDeClasse ();

NomDeClasse objet2 = objet1;

objet1 et objet2 contiennent la même référence et pointent donc tous les deux sur le même espace mémoire : les modifications faites à partir de l'une des deux variables modifient le contenu aussi bien pour l'une que pour l'autre des 2 variables.

2.7 Références et comparaison d'objets

Les variables de type objet que l'on déclare ne contiennent pas un objet mais une référence vers cet objet. Lorsque l'on écrit `c1 = c2` (`c1` et `c2` sont des objets), on copie la référence de l'objet `c2` dans `c1` : `c1` et `c2` réfèrent donc le même objet (ils pointent sur le même objet). L'opérateur `==` compare ces références. Deux objets avec des propriétés identiques sont deux objets distincts :

Exemple :

```
Point p1 = new Point (100,50);
```

```
Point p2 = new Point (100,50);
```

```
if (p1 == p2) { ... } // Faux
```

```
if (p1.equals(p2)) { ... } // Vrai
```

Pour tester l'égalité des contenus de deux instances, il faut munir la classe d'une méthode permettant de réaliser cette opération : la méthode `equals` héritée de `Object`.

Remarque :

Pour s'assurer que deux objets sont de la même classe, il faut utiliser la méthode `getClass()` de la classe `Object` dont toutes les classes héritent.

Exemple :

```
(obj1.getClass().equals(obj2.getClass()))
```

2.8 Le mot clé this

Ce mot clé sert à référencer dans une méthode l'instance de l'objet en cours d'utilisation. `this` est un objet qui est égale à l'instance de l'objet dans lequel il est utilisé.

Ainsi dans une classe, pour accéder à une propriété « `p` », on peut écrire `this.p`. cette notation devient nécessaire dans le cas où une autre variable du contexte (un paramètre ou une variable globale) porte le même nom de la propriété.

Exemple :

```
class A {  
    private int value;  
    public A(int value) {  
        value = value;  
        this.value = value;  
    }  
}
```

Cette référence est habituellement implicite :

Exemple :

```
class B {  
    private String text = "abc" ;  
    public String getText() {  
        return text;  
    }  
}
```

2.9 L'opérateur instanceof

L'opérateur instanceof permet de déterminer la classe de l'objet qui lui est passé en paramètre. La syntaxe est :

... objet **instanceof** classe ...

Exemple :

```
void testClasse(Object x) {  
    if (x instanceof A ) ...  
}
```

2.10 Les constantes

Les constantes sont définies avec le mot clé **final** : leur valeur ne peut pas être modifiée. Elle sont généralement aussi : publique et statiques.

Exemple :

```
public class A {  
    public static final double PI = 3.14 ;  
}
```

2.11 Surcharge (ou Surdéfinition) des méthodes

La surcharge d'une méthode permet de définir plusieurs fois une même méthode avec des arguments différents. Lors de l'appel à la méthode, le compilateur choisi la méthode qui doit être appelée en fonction du nombre et du type des arguments.

Une méthode est surchargée lorsqu'elle exécute des actions différentes selon le type et le nombre de paramètres transmis.

```
public class Number {  
    private int value;  
    public Number(int value) {  
        This.value = value;  
    }  
    public void inc() {  
        value ++;  
    }  
    public void inc(int step) {  
        value = value + step;  
    }  
}
```

```
Number n = new Number(20) ;  
n.inc();    //appellera la première méthode inc() et incrémentera la valeur par  
1  
n.inc(5);   //appellera la 2ème méthode inc(int) et incrémentera la valeur par  
5
```

Remarque :

Les constructeurs peuvent aussi être surchargés

2.12 Les accesseurs

L'encapsulation permet de sécuriser l'accès aux données d'une classe. Ainsi, les données déclarées private à l'intérieur d'une classe ne peuvent être accédées et modifiées que par des méthodes définies dans la même classe. Si une autre

classe veut accéder aux données de la classe, l'opération n'est possible que par l'intermédiaire de méthodes de la classe prévues à cet effet.

Un accesseur est une méthode publique qui donne l'accès à une propriété privée d'une classe. L'accès peut être réalisé en lecture ou en écriture. Par convention, les accesseurs en lecture commencent par le mot get (ou is ce le type de la propriété est boolean) et sont appelés des getters et les accesseurs en écriture commencent par le mot set et sont appelés des setters.

```
public class Number {  
    private int value;  
    public Number(int value) {  
        This.value = value;  
    }  
    public void setValue(int value) {  
        this.value = value;  
    }  
    public int getValue() {  
        return value;  
    }  
}
```

2.13 Définition d'un package

En java, il existe un moyen de regrouper des classe voisines ou qui couvrent un même domaine dans une même structure appelée : package. Pour réaliser un package, on écrit un nombre quelconque de classes dans plusieurs fichiers d'un même repertoire et on commence le fichier de classe avec l'instruction suivante :

```
package nom.de.package;
```

Le mot clé package doit être la première instruction dans un fichier source et il ne doit être présent qu'une seule fois dans le fichier source (une classe ne peut pas appartenir à plusieurs packages).

La hiérarchie d'un package se retrouve dans l'arborescence du disque dur puisque chaque package est dans un répertoire nommé du nom du package.

2.14 Utilisation d'un package

Pour utiliser ensuite le package ainsi créé, on l'importe dans le fichier :

```
import nom.de.package.*;
```

Pour utiliser une classe A particulière du package, on peut utiliser l'instruction suivante :

```
import nom.de.package.A;
```

Remarque :

L'utilisation du symbole * donne l'accès à toutes les classes du package mais pas aux sous packages de celui-ci.

Chapitre 3. Éléments de base du langage Java

3.1 Variables et Objets

En Java, il existe 3 catégories de types :

1- les types de base (types primitifs)

2- le type Classe

3- le type tableau

Les types de base (en nombre de 8 plus le type void) sont les seuls types statiques (non dynamiques). Les Objets et les tableaux sont toujours et implicitement des données dynamiques créées par l'intermédiaire du mot clé **new**.

3.2 Types de base

le langage Java offre 8 types de base appelés aussi primitives ou **types primitifs**. Le mot clé **void** n'est plus un type, il est utilisé uniquement pour définir les procédures,

Type	Désignation	Plage de valeurs
byte	Entier signé 1 octet	-128 → 127
short	Entier signé 2 octets	-32768 → 32767
int	Entier signé 4 octets	$-2^{31} = -2147483648 \rightarrow 2^{31}-1 = 2147483647$
long	Entier signé 8 octets	$-2^{63} = -9223372036854775808 \rightarrow 2^{63}-1 =$

	octets	9223372036854775807
float	Réel sur 4 octets	
double	Réel sur 8 octets	$-3.4 \times 10^{38} \rightarrow 3.4 \times 10^{38}$
char	Caractère 2 octets	Code en Unicode: char c = 'x'; OU char c = '\u0058';
boolean	Type logique	true et false

Exemple :

```
int x = 200 ;
```

```
byte b = 64 ;
```

3.3 Classes et Objets

Création :

La création des objets est toujours réalisée par l'intermédiaire de l'opérateur new :

```
NomDeClasse objet;  
objet = new NomDeClasse(paramètres d'un constructeur);
```

ou encore :

```
NomDeClasse objet = new NomDeClasse(paramètres d'un constructeur);
```

Exemple :

```
String S = new String("ceci est une chaîne de caractères");
```

Copie

Un objet peut aussi référencer l'adresse d'un objet existant. Les deux objets désigneront alors la même information. Cependant la destruction de l'un des objets ne causera pas la destruction de l'autre.

```
NomDeClasse objet1 = new NomDeClasse(paramètres d'un constructeur);  
NomDeClasse objet2 = objet1;
```


Exemple :

```
String S1 = new String("ceci est une chaîne de caractères");  
String S2 = S1;
```

Les chaînes de caractères constituent un cas particulier. Elles peuvent en plus être initialisées à des constantes chaînes :

```
S2 = "abc"
```

Destruction

Un objet en mémoire est détruit automatiquement par l'intermédiaire du Garbage Collector. Aucune destruction explicite n'est alors nécessaire. Cependant, la constante null peut être affectée à un objet pour supprimer le lien avec l'espace mémoire qu'il référençait auparavant. Ce qui entraînera la libération de la mémoire si celle-ci n'est pas référencée par un autre objet.

```
objet = null;
```

3.5 Quelques propriétés du Langage Java

1. Les tableaux tels qu'ils sont définis en Java (objets dynamiques) ne peuvent pas remplacer la notion de liste. En effet, dès qu'ils sont créés, on ne peut plus leur modifier la taille, sauf si on remplace le tableau créé par un autre.
2. Pour gérer les listes, on peut :
 - Les créer à l'aide de classes.
 - Ou encore utiliser une classe Liste existante : LinkedList , Vector ou Stack du package java.util.
3. Un tableau de char n'est pas une chaîne de caractères en Java. Il ne peut donc pas être géré comme une chaîne.

Exemple :

L'écriture :

```
char S[] = "abc" ;
```

est incorrecte.

4. Il est possible de convertir un tableau char [] en une chaîne (String) par l'intermédiaire de l'un des constructeurs de la classe String.

Exemple :

```
char S1[] = {'a','b','c'};
```

```
String S2 = new String(S1) ;
```

5. Les types wrappers tels que **Integer** sont en principe utilisés pour permettre la conversion automatique vers la classe mère **Object**. Ce qui n'est pas le cas pour les types de base (**int**, **float**, ...).

Exemple :

```
Vector L = new Vector();
```

```
int x=20;
```

```
Integer wx=new Integer(20);
```

L.add(**x**); // est incorrecte avant la version 5 (avec celle-ci, cela devient

// possible à l'aide de la notion d'**autoboxing**) alors que

```
L.add(wx); // est correcte.
```

6. Les variables de types élémentaires (primitives) sont manipulées directement par valeur tandis que tous les autres objets sont manipulés par leurs adresses. Ces derniers nécessitent alors une création explicite après leur déclaration (par l'intermédiaire de l'opérateur **new**).

7. Un résultat de la remarque précédente est qu'au niveau des procédures et fonctions le passage des paramètres est toujours par adresse pour le cas des objets. Quant aux paramètres de type de base, le passage est par valeur. Afin de réaliser un passage par adresse pour ces derniers on peut les envelopper dans des objets de classe que l'on crée pour cette fin.

Exemple :

```
class Entier {  
    int val ;  
    public int getVal() {return val ;}  
    public void setVal(int v) {val = v ;}  
}
```

Un paramètre de type **Entier** utilisera alors un passage par adresse, à l'inverse d'un paramètre de type **int**.

8. Les propriétés **static** sont accessibles directement par l'intermédiaire du nom de la classe : `NomDeClasse.propriété`
9. Les méthodes **static** ne peuvent accéder qu'aux propriétés et méthodes **static** (ou à des variables locales). Par conséquent, la méthode **main** (qui est static) ne peut pas accéder à des propriétés non static (ni appeler des méthodes non static).

Exemple 1 :

```
class Class1 {  
    private static int x ;  
    private int y ;  
    public static void main(String args[]) {  
        x = 20 ;    // correcte  
        y = 30 ;    // incorrecte  
        ...  
    }  
}
```

Exemple 2 :

```
class Class1 {  
    private int x, y ;  
    void initData() {  
        x = 20 ;  
        y = 30 ;  
    }  
    public static void main(String args[])  
    {  
        initData() ;    // incorrecte  
    }  
}
```

Une solution à ce problème peut être obtenue, on ajoutant un constructeur à la classe qui appelle la procédure initData(), et de créer une instance de la classe dans le main :

```
class Class1 {  
    private int x, y ;  
    Class1() {  
        initData() ;  
    }  
    void initData() {  
        x = 20 ;  
        y = 30 ;  
    }  
    public static void main(String args[]) {  
        new Class1() ;  
    }  
}
```

10. Les variables de classes (ou propriétés) peuvent être initialisées (à l'inverse du C++) lors de leur déclaration.

Exemple :

```
class Class1 {  
    private int x=20, y=30 ;  
    ...  
}
```

11. Toutes les classes prédéfinies ou définies par l'utilisateur lui même dérivent directement ou indirectement de la classe **Object** (**java.lang.Object**). En conséquence, Les méthodes qui acceptent des paramètres de type Object peuvent recevoir des arguments instances de n'importe quelle classe.

12. Pour déterminer le nom de classe d'un objet, on peut utiliser dans l'une des méthodes (non statics) de la classe l'instruction : `getClass().getName()`. La méthode `getClass()` retourne la classe (un objet de type **Class**) et la méthode `getName()` (de la classe **Class**) retourne, sous forme de chaîne de caractères, le nom de la classe :

```
System.out.println("je suis un objet de la classe : " +  
                    getClass().getName());
```

13. Pour déterminer le nom de classe mère d'un objet, on peut utiliser dans l'une des méthodes (non statics) de la classe l'instruction :

`getClass().getSuperclass().getName()`.

La méthode `getSuperclass()` de la classe **Class** retourne un objet de type **Class** qui contient la classe mère de l'objet en cours :

```
System.out.println("Ma classe mère : "  
+getClass().getSuperclass().getName());
```

14. Pour déterminer le nom de la classe mère d'une classe dont on connaît le nom, on peut utiliser l'instruction :

`Class.forName("Nom de la classe").getSuperclass().getName()`

Exemple :

```
try {  
    System.out.println("mère de java.lang.Thread :" +  
        Class.forName("java.lang.Thread").getSuperclass().getName());  
} catch (Exception e) {}
```

15. La mémoire occupée par un objet est automatiquement libérée lorsque celle-ci n'est plus référencée par aucune variable du programme. Cette désallocation automatique est réalisée par un programme de la machine virtuelle Java appelé : **Garbage Collector**. Celui-ci peut cependant être appelé explicitement par l'intermédiaire de la méthode static `gc()` de la classe **System**.

3.5 La classe Object

L'héritage est un concept qui est toujours utilisé en Java. Ainsi à chaque fois que l'on définit une nouvelle classe, celle-ci se dérive automatiquement de la classe **Object** mère de toutes les classes :

```
class Test {  
    Test() {  
        String S = getClass().getName();  
        System.out.println("Je suis un nouvel objet de la classe : "+S);  
    }  
    public static void main(String args[]) {  
        new Test();  
    }  
}
```

A l'exécution, le programme affiche le message :

Je suis un nouvel objet de la classe : Test

getClass étant une méthode de la classe **Object** retournant un objet de type Class qui représente la classe de l'objet appelant. La méthode **getName()** de la classe Class permet d'obtenir le nom de la classe sous forme d'une chaîne de caractère.

Voici la liste des méthodes de la classe **Object** :

1- protected Object clone()

Créer une copie de l'objet appelant.

2- boolean equals(Object obj)

Teste si l'objet obj est égal à l'objet appelant. La fonction retourne true si les objet ont la même référence. Exemple :

<pre>Class1 v1 = new Class1(); Class1 v2 = v1; if (v1.equals(v2)) ...</pre>

La méthode equals peut être redéfinie pour faire une comparaison sur le contenu et non la référence.

3- protected void finalize()

Cette méthode est appelée par le Garbage Collector lorsqu'il veut éliminer un objet de la mémoire. Si l'on veut réaliser un traitement particulier il suffit de redéfinir cette méthode avec le comportement adéquat dans les classes utilisées.
--

4- Class getClass()

Retourne un objet de type Class qui représente la classe à laquelle appartient l'objet appelant.
--

Pour afficher le nom de cette classe, il suffit d'appeler la méthode getName() sur l'objet retourné :
--


```
String S = getClass() .getName() ;
```

Pour créer un nouvel objet de cette même classe, on utilise la méthode **newInstance()** :

```
Class c = getClass() ;  
try {  
    c.newInstance() ;  
} catch (Exception e) {}
```

5- int hashCode()

Retourne un Hash-Code pour l'objet appelant.

6- void notify()

Cette méthode est appelée uniquement dans un moniteur (dans une méthode **synchronized**) permettant de réveiller l'un d'entre les processus qui se sont fait bloquer dans ce même moniteur. Le choix du processus à réveiller est réalisé par le système. L'objet appelant ignore complètement quel sera le prochain processus réveillé.

7- void notifyAll()

Comme notify(), mais réveille tous les processus bloqués dans le moniteur.

8- String toString()

Retourne une représentation sous forme de String de l'objet, affichable par l'intermédiaire de la méthode System.out.println(). La méthode toString() est appelée à chaque fois que l'on essaie d'afficher un objet par la méthode System.out.println(). Il suffit donc de la redéfinir dans la classe de l'objet avec le comportement adéquat. Exemple :

```
class Point {  
    int x, y;  
    ...  
    public String toString() {  
        return "(" + x + " , " + y + ")";  
    }  
}
```

9- void wait ()
Cette méthode est appelée, comme notify(), à l'intérieur d'un moniteur permettant le blocage du processus appelant qui restera bloqué jusqu'à recevoir un signal notify() envoyé par un autre processus.
10- void wait (long timeout)
Comme wait() mais le blocage est pendant timeout millisecondes
11- void wait (long timeout, int nanos)
Comme wait() mais le blocage est pendant timeout millisecondes et nanos nanosecondes

Remarque :

Puisque toutes les classes Java (existantes ou créées par l'utilisateur) descendent de la classe Object, si on a alors une méthode qui reçoit en paramètre une variable de type Object, on peut alors communiquer à cette méthode un objet de n'importe quel type. Comme exemple, on peut citer la méthode add de la classe **LinkedList** (Liste chaînée) du package java.util. Celle-ci accepte en paramètre une variable de type Object (**add(Object o)**). On peut donc remplir la liste avec des objets de types variés :

```
LinkedList L = new LinkedList();
L.add(new String("abc") );
L.add(new Integer(20) );
```

3.6 Gestion des exceptions en Java

La gestion des exceptions ou erreurs en Java offre des moyens structurés pour capturer les erreurs d'exécution d'un programme et de fournir des informations significatives à leur sujet. Pour le traitement des exceptions, on utilise les mots clés **try**, **catch** et **finally** :

```
try {  
    // Code pouvant se terminer en erreur et déclencher une exception.  
}  
catch( Exception e ) {  
    // Code de traitement de l'exception.  
    // La ligne suivante ouvre un suivi de pile de l'exception :  
    e.printStackTrace();  
}  
finally {  
    // Le code inséré ici sera toujours exécuté,  
    // que l'exception ait été déclenchée dans le bloc try ou non.  
}
```

- 1- Le bloc **try** doit être utilisé pour entourer tout code susceptible de déclencher une exception ayant besoin d'être gérée. Si aucune exception n'est déclenchée, tout le code du bloc try est exécuté. Mais, si une exception est déclenchée, le code du bloc try arrête l'exécution à l'endroit où l'exception a été déclenchée et le contrôle passe au bloc catch, dans lequel l'exception est gérée.
- 2- Le bloc **catch** permet de traiter l'exception. On peut faire tout ce dont on a besoin pour gérer l'exception dans un ou plusieurs blocs catch. Le moyen le plus simple de gérer des exceptions peut être réalisé à travers un seul bloc catch. Pour cela, l'argument entre les parenthèses suivant catch doit indiquer la classe Exception, suivie d'un nom de variable à affecter à cette exception. Cela indique que toute exception qui est une instance de

java.lang.Exception ou de n'importe laquelle de ses sous-classes sera capturée.

- 3- Le bloc **finally** est optionnel. Le code se trouvant dans le bloc finally sera toujours exécuté, même si le bloc try qu'il déclenche une exception et ne se termine. Le bloc finally est un bon endroit pour placer du code de nettoyage.

Chapitre 4. Héritage Java

1. Une classe B **étend** une class A pour **ajouter ou modifier des services**. On peut aussi ajouter des propriétés.
2. La nouvelle classe B est appelée : classe **Fille**, ou classe **Dérivée**.
3. La classe A est la classe **Mère** ou **classe de Base**.
4. On dit que **B étend A** ou **B hérite de A**.

Contraintes :

5. Si la classe A ne dispose d'aucun constructeur (→ le compilateur génère un constructeur sans paramètre pour A), la classe B peut être définie sans contraintes.

Exemple :

```
class A {  
    private int x, y;  
  
    public void setX(int x) {  
        this.x = x;  
    }  
  
    public int getX() {  
        return x;  
    }  
}
```

```
public int getY() {  
    return y;  
}  
  
public void print() {  
    System.out.println("(" + x + ", " + y + ")");  
}  
}
```

```
Class B extends A {  
  
  
}
```

➔ **Un constructeur sans paramètre est généré pour B.**

6. Si A dispose d'un constructeur avec paramètres et pas de constructeur sans paramètres, alors la définition suivante est erronée :

```
Class B extends A {  
  
  
}
```

7. En fait un constructeur sans paramètre est généré pour B, celui-ci appellera le constructeur sans paramètre de A qui n'existe pas. D'où l'erreur

8. **Règle Générale :** tous les constructeurs d'une classe Fille appellent automatiquement le constructeur sans paramètre de la classe Mère. Cet appel est généré automatiquement par le compilateur comme première instruction de chaque constructeur.
9. Cet appel au constructeur de la classe Mère **peut** être réalisé explicitement à l'aide du mot clé **super** :

```
Class B extends A {  
    B() {  
        super() ;  
        ...  
    }  
}
```

10. Cet appel comme première instruction Signifie que si on a une class C qui dérive de B, alors :
- Le constructeur de A est exécuté en premier lieu
 - Ensuite celui de B
 - Et enfin celui de C :

```
Class A {  
  
    A() {  
  
        System.out.println("A");  
  
    }  
  
}
```

```
Class B extends A {  
  
    B() {  
  
        System.out.println("B");  
  
    }  
  
}
```

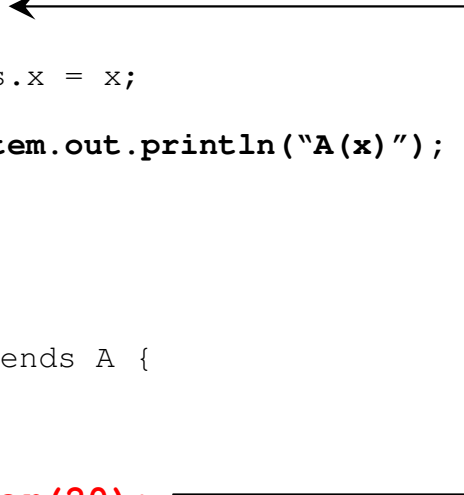
```
Class C extends B{  
  
    C() {  
  
        System.out.println("C");  
  
    }  
  
}
```

Une instantiation de la classe C → l'affichage suivant:

```
A  
  
B  
  
C
```


11. L'existence de ce mot clé **super** permettra de dérouter l'appel à un autre constructeur avec paramètres en précisant entre les parenthèses les arguments d'appel de celui-ci :

```
class A {  
    private int x;  
  
    A() {  
        System.out.println("A");  
    }  
  
    A(int x) ←  
        This.x = x;  
        System.out.println("A(x)");  
    }  
}  
  
Class B extends A {  
    B() {  
        super(20); _____  
        System.out.println("B");  
    }  
}
```



Au résultat, on aura:

A(x)

B

12. Chaque super appelle la classe Mère immédiatement supérieure



```
class A {  
    private int x;  
  
    A() {  
        System.out.println("A");  
    }  
  
    A(int x) { ←  
        This.x = x;  
        System.out.println("A(x)");  
    }  
}
```

```
Class B extends A {  
    B() {  
        super(20);  
        System.out.println("B");  
    }  
}
```



```
Class C extends B {  
    C() {  
        super(20); // est une erreur  
        System.out.println("B");  
    }  
}
```

Signifie que **B** a un constructeur avec un paramètre entier, ce qui n'est pas le cas.

13. **Remarque :** Un constructeur d'une classe peut aussi appeler un autre constructeur de la même classe (pour réutiliser les instructions définies dans ce dernier). On utilisera le mot clé **this** :

```
class A {  
    private int x;  
  
    A() {    
        System.out.println("A");  
    }  
  
    A(int x) {  
        this();   
  
        this.x = x;  
  
        System.out.println("A(x)");  
    }  
}
```

Ou encore :

```
class A {  
    private int x;  
  
    A() {  
        this(20);   
        System.out.println("A");  
    }  
  
    A(int x) {   
        this.x = x;  
  
        System.out.println("A(x)");  
    }  
}
```

14. L'instruction **this(...)** doit aussi être la toute première instruction du constructeur.

15. L'extension d'une classe permettra d'ajouter des services :

```
Class A {  
    A() {  
        System.out.println("A");  
    }  
    void s1() {  
        ...  
    }  
    void s2() {  
        ...  
    }  
}
```

```
Class B extends A {  
    B() {  
        System.out.println("B");  
    }  
    void s3() {  
        ...  
    }  
}
```

Une instance « a » de « A » aura droit aux services : s1() et s2()

*Une instance « b » de « B » aura droit aux services :
s1(), s2() (hérités) ainsi qu'au service s3().*

C'est à dire :

```
A a = new A() ;  
B b = new B() ;  
a.s1(); a.s2(); a.s3();  
b.s1(); b.s2(); b.s3();
```

16. L'extension d'une classe permettra de modifier des services. On parlera de la **Redéfinition** :

```
Class A {  
    A() {  
        System.out.println("A");  
    }  
    void s1() {  
        System.out.println("Service s1() de A");  
    }  
}
```

```
Class B extends A {  
  
    B() {  
  
        System.out.println("B");  
  
    }  
  
    void s1() {  
  
        System.out.println("Service s1() Redéfini dans B");  
  
    }  
  
}
```

```
A a = new A() ;  
  
B b = new B() ;  
  
a.s1(); ➔ "Service s1() de A";  
b.s1(); ➔ "Service s1() Redéfini dans B";
```

17. Si on veut réutiliser le service `s1()` de la classe Mère dans la classe Fille, il est possible d'appeler le service `s1()` de la classe Mère depuis la Fille à l'aide de l'instruction **super.s1()** :

```
Class B extends A {  
  
    B() {  
  
        System.out.println("B");  
  
    }  
  
    void s1() {  
  
        System.out.println("Service s1() Redéfini dans B");  
        super.s1();  
  
    }  
  
}
```

```
B b = new B() ;  
  
b.s1(); ➔      "Service s1() Redéfini dans B";  
               "Service s1() de A";
```

En résumé : Le mot clé **super** désigne la classe mère. Il peut être utilisé dans deux situations différentes :

- 1- Pour appeler un constructeur de la classe mère : il est alors utilisé comme **première** instruction du **constructeur** de la classe fille :

Syntaxe :

```
super(paramètres du constructeur de la classe mère);
```

Exemple :

Classe Mère :

```
class A {  
    private String name;  
    A(String S) {  
        name = S;  
    }  
    String getName() {  
        return name;  
    }  
}
```

Classe Fille :

```
class B extends A {  
    B(String Nom) {  
        super(Nom);  
    }  
    public static void main(String[] args) {  
        B objet = new B("objet B");  
        System.out.println(B.getName());  
    }  
}
```

- 2- Pour appeler une méthode de la classe mère, si celle-ci est redéfinie dans la classe fille :

Syntaxe :

```
super.nomDeLaMethode(paramètres);
```

Exemple :

```
class A {  
    void p()  
    {  
        System.out.println("Méthode p() de la classe A");  
    }  
}
```



```

public class B extends A {
    void p() {
        System.out.println("Méthode p() de la classe B");
        super.p();
    }
    public static void main(String[] args) {
        B objet = new B();
        objet.p();
    }
}

```

Le programme affiche :

```

Méthode p() de la classe B
Méthode p() de la classe A

```

18.Remarque : la classe **Object** est la classe mère de toutes les classes sans Mère. Ainsi la définition suivante :

```

class A {
    ...
}

```

Est équivalente à :

```

class A extends Object {
    ...
}

```

Résultats :

19. **Résultat N° 1 :** La classe **Object** se trouve en tête de l'arbre d'héritage de toutes les classes Java.
20. **Résultat N° 2 :** Toute nouvelle classe va hériter de toutes les méthodes « public » de la classe Object.
21. **Résultat N° 3 :** comme exemple de méthodes, il y a la méthode **toString()** invoquée automatiquement par le compilateur lors d'une tentative d'affichage d'un objet :

```
System.out.println(objet)
```

⇔

```
System.out.println(objet.toString()) //générée par le compilateur
```

Mais aussi dans tout contexte nécessitant cette transformation :
généralement les concaténation de chaînes de caractères :

```
String s ;  
s + objet + ... ⇔ s + objet.toString() + ...
```

22. la méthode `toString()` est une méthode public de la classe Object qui retourne une chaîne de caractère ; l'objet est alors « imprimable » ou « affichable ».
23. La méthode `toString()` est définie par défaut (dans Object) pour retourner l'adresse de l'objet sous forme d'une chaîne de caractères.
24. Il est donc possible de la redéfinir dans n'importe quelle classe pour choisir le format adéquat d'afficher des objets de la classe.

Exemple :

```
class A {  
    private int x, y;  
    ...  
    public String toString() {  
        return "(" + x + ", " + y + ")";  
    }  
}
```

NECESSAIRE

25. La redéfinition d'une méthode ne doit pas réduire ses privilèges d'accès :

```
private void p() ; → (Red) → private, protected, "sans" ,  
public  
protected void p() ; → (Red) → protected, "sans" , public  
void p() ; → (Red) → "sans" , public  
public void p() ; → (Red) → public
```

26. Important : un objet de classe Fille peut être affecté à un objet de classe Mère :

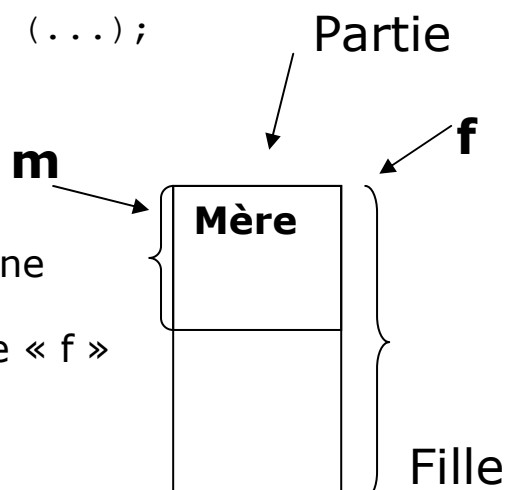
L'affectation [**Mère ← Fille**] est possible :

```
Fille f = new Fille (...);
```

```
Mère m = f ;
```

→ l'objet « m » aura une

Vue plus restreinte que « f »



27. Important : L'affectation [Mère → Fille] n'est possible que :

- Si on utilise un **transtypage** (Casting)
- Si l'objet Mère pointe effectivement un Objet Fille :

Exemple :

```
Fille f1 = new Fille(...) ;  
  
Mère m = f1 ; // l'objet mère Pointe une fille.  
  
Fille f2 = (Fille) m ; // Possible
```

```
Mère m2 = new Mère(...) ;  
  
Fille f3 = (Fille)m2 ;// Possible lors de la compilation (syntaxiquement).  
  
Mais ERREUR lors de l'Exécution.
```

Remarques :

- 28. Le casting des objets en Java permet de Retrouver l'identité réelle d'un objet.
- 29. Pas de casting entre de objets de classes n'ayant pas de relation Mère-Fille
- 30. Cette relation peut ne pas Entre directe :

Exemple :

C étend B et B étend A, on peut faire :

```
C c1 = new C(...) ;  
  
A a = c1 ; // Mère ← Fille  
  
C c2 = (C) a; // Fille ← Mère
```

1.5 Classes Abstraites et Interfaces

En Java, il existe 3 types d'entités qu'on peut manipuler :

1. Les **classes** (déjà vues)
2. Les **classes abstraites** présentées par le mot clé **abstract** :

```
abstract class NomClasse {  
  
    ...  
  
}
```

Dans une classe abstraite, le corps de quelques méthodes peut ne pas être défini (on déclare uniquement le prototype de la méthode). Ces méthodes sont dites des méthodes abstraites. Une méthode abstraite est aussi présentée par l'intermédiaire du mot clé **abstract** de la manière ci-après. C'est aux classes dérivées de redéfinir ces méthodes et de préciser leur comportement.

```
abstract class NomClasse {  
  
    abstract type NomMéthode(parameters);  
  
    ...  
  
}
```

Une classe abstraite ne peut donc jamais être instanciée. Il s'agit d'une spécification devant être implémentée par l'intermédiaire d'une classe dérivée. Si cette dernière définit toutes les méthodes abstraites alors celle-ci est instanciable.

Remarque :

Une classe abstraite (présentée par le mot clé **abstract**) peut ne pas contenir de méthodes abstraites. Cependant, une classe contenant une méthode abstraite doit obligatoirement être déclarée **abstract**.

3. Les **interfaces** qui sont définies par l'intermédiaire du mot clé **interface** au lieu de **class** constituent un cas particulier des classes abstraites : d'une part, ce sont des *classes* où aucune méthode n'est définie (uniquement le prototype de chaque méthode). D'autre part, l'extension d'une interface est appelée **implémentation** et elle est réalisée par l'intermédiaire du mot clé **implements** :

- Définition d'une interface :

```
interface NomInterface {  
    type1 methode1(paramètres);  
    type2 methode2(paramètres);  
    ...  
}
```

- Implémentation d'une interface :

```
class NomDeClasse implements NomInterface {  
    ...  
}
```

Remarques :

1. Une classe qui implémente une interface doit définir toutes les méthodes de l'interface.
2. Une interface peut aussi contenir des attributs.
3. Tous les attributs d'une interface doivent obligatoirement être initialisés.
4. Tous les attributs d'une interface sont **public**, **static** et **final**.

Chapitre 5. Les Tableaux

5.1 Déclaration :

```
Type NomDuTableau[] ;
```

Avec Type est l'un des deux cas suivant :

- un type primitif
- une classe

Exemples :

```
int T1[] ;  
  
Vector T2[] ;  
  
String T3[] ;
```

5.2 Création du Tableau

```
NomDuTableau = new Type[taille] ;
```

Exemples :

```
T1 = new int[20] ;  
  
T2 = new Vector[10] ;  
  
T3 = new String[15] ;
```

Remarque :

Dans le cas des tableaux de classes, seul le tableau est créé mais pas les éléments ce qui nécessite pour le cas de T2 et T3 la création de chaque T2[i] et chaque T3[j]

5.3 Création des éléments d'un tableau

- Pour T1 (**tableau de primitifs**) :

T1[0] = 34 ;

T1[1] = 45 ;

...

- Pour T2 et T3 (**tableaux d'objets**) : **les éléments doivent être créés avant de s'en servir**

T2[0] = new Vector() ; T2[0].add(...) ; ...

T3[0] = new String("....") ; if (T3[0].equals(...) ...

5.4 Le tableau est un Objet

➔ Il dispose des méthodes héritées de la classe Object, en plus il dispose de la propriété à lecture seule : **length**

```
int l = T1.length ; ➔ 20
```

Remarque1 :

Lorsqu'un tableau est créé, il est impossible de le redimensionner.

La solution :

- créer un nouveau tableau avec la taille souhaitée
- copier les éléments du tableau initial
- supprimer ce tableau

Exemple :

```
int T1[] = new int[20] ;  
  
int tmp[] = new int[21] ;  
  
for (int i=0 ; i<T1.length ; i++) tmp[i] = T1[i] ;  
  
tmp[20] = nouvelle valeur ;  
  
T1 = tmp ;
```


Remarque21 :

1. Les 2 écritures suivantes sont équivalentes :

```
Type T[] ;
```

⇔

```
Type []T ;
```

2. Une fonction peut retourner un tableau:

```
int []nomDeMethode(...) {  
  
    ...  
  
}
```

5.5 Initialisation des tableaux

Un tableau peut être initialisé lors de sa déclaration :

1. Cas d'un tableau de primitifs :

```
int T1[] = {3, 5, 7, 34} ;
```

⇔

```
int T1[] = new int[4] ;  
  
T1[0] = 3 ;  
  
T1[1] = 5 ;  
  
T1[2] = 7 ;  
  
T1[3] = 34 ;
```

2. Cas d'un tableau d'objets :

1^{ère} Solution :

```
Vector v1 = new Vector() ;  
  
Vector v2 = new Vector() ;  
  
...  
  
Vector T2[] = {v1, v2, ...} ;
```

2^{ème} Solution (pratique) :

```
Vector T2[] = {  
    new Vector(), // ce sont des objets créés à la volée (objets anonymes)  
    new Vector(),  
    ...  
} ;
```

5.6 Création de tableaux anonymes :

Soit une méthode ayant un tableau comme paramètre :

```
void p1(int T[]) {  
  
    ...  
  
}  
  
...
```

Appeler la méthode :

1^{ère} solution :

```
int T1[] = {2, 6, 17} ;  
p1(T1) ;
```

2^{ème} Solution : tableau anonyme

```
p1( new int[] {2, 6, 17} ) ;
```

Syntaxe générale :

```
... new type [] { valeur/objet, valeur/objet, ...} ...
```

5.7 Tableau à plusieurs dimensions

Type nomDuTableau[][]... ;

Exemple :

```
int M[][] ;
```

Création :

```
M = new int[3][10] ;
```

Accès :

```
M[i][j] = ..
```

Remarque :

Si on suppose que la 1^{ère} dimension. C'est les lignes et la 2^{ème} dimension.
C'est les colonnes :

M.length : le nombre de lignes

M[i].length : le nombre de colonne de la ligne i

Initialisation :

```
int M [][] = {  
    {2, 3, 4},  
    {1, 45, 6}  
};
```

→ Chaque M[i] est un tableau :

M[0] → {2, 3, 4}

M[1] → {1, 45, 6}

→ M.length = 2 et

M[0].length = M[1].length = 3

Résultat :

Chaque ligne de la matrice peut avoir un nombre d'élément différent:

Exemple :

```
int M [][] = {  
    {2, 3},  
    {1, 45, 6, 3},  
    {3, 7, 9}  
};
```

Remarques :

1. Une matrice M est un tableau de tableaux
2. Chaque M[i] est un tableau pouvant être créé séparément
1. La matrice M peut être créée comme un tableau dont les éléments sont créés ultérieurement

Exemple :

```
int M[][] ;  
  
M = new int[3][] ;  
  
...  
  
M[0] = new int[2] ;  
M[1] = new int[4] ;  
M[2] = new int[3] ;  
  
...
```

Chapitre 6. Les Fichiers

6.1 Les flux en JAVA

Le Package **java.io** offre une véritable collection de classes permettant la gestion des Entrées/Sortie. On énumère alors les classes suivantes.

BufferedInputStream	FilterInputStream	PipedOutputStream
BufferedOutputStream	FilterOutputStream	PipedReader
BufferedReader	FilterReader	PipedWriter
BufferedWriter	FilterWriter	PrintStream
ByteArrayInputStream	InputStream	PrintWriter
ByteArrayOutputStream	InputStreamReader	PushbackInputStream
CharArrayReader	LineNumberInputStream	PushbackReader
CharArrayWriter	LineNumberReader	RandomAccessFile
DataInputStream	ObjectInputStream	Reader
DataOutputStream	ObjectInputStream.GetField	SequenceInputStream
File	ObjectOutputStream	SerializablePermission
FileDescriptor	ObjectOutputStream.PutField	StreamTokenizer
FileInputStream	ObjectStreamClass	StringBufferInputStream
FileOutputStream	ObjectStreamField	StringReader
FilePermission	OutputStream	StringWriter
FileReader	OutputStreamWriter	Writer
FileWriter	PipedInputStream	

6.2 Fichiers Textes

Fonctions de lecture : classe **FileReader**

Classe utilisée :

`FileReader`

Constructeur :

`FileReader` ([`String`](#) NomDeFichier)

→ `FileReader f = new FileReader("test.txt");`

Si le fichier n'existe pas l'ouverture déclenche une exception qui peut être attrapée de la manière suivante :

```
try {  
    f = new FileReader("test2.dat");  
}  
catch (Exception e) {  
    System.out.println("Erreur d'ouverture du fichier");  
    System.out.println(e.getMessage());  
    System.exit(0);  
}
```

Méthodes :

<code>int</code> <code>read()</code> .	Lit un caractère. Retourne -1 si fin de fichier.
<code>int</code> <code>read(char[] cbuf)</code>	Lit un tableau de caractères et retourne le nombre de caractères lus
<code>int</code> <code>read(char[] Tchar, int debut, int Nombre)</code>	lit une séquence de caractères dans une portion de tableau et retourne le nombre de caractères lus.

Exemple 1 : Lecture d'un fichier texte caractère par caractère

```
import java.io.*;
class ES01 {
    public static void main(String args[]) throws IOException
    {
        FileReader f=null;
        try {
            f = new FileReader("test.txt");
        }
        catch (IOException e) {
            System.out.println("Erreur d'ouverture du fichier");
            System.out.println(e.getMessage());
            System.exit(0);
        }
        int c;
        while ((c=f.read())!=-1) System.out.print((char)c);
        f.close();
    }
}
```


Exemple 2 : Lecture d'un fichier texte tampon par tampon

```
import java.io.*;
class ES02 {
    public static void main(String args[]) throws IOException
    {
        FileReader f=null;
        try {
            f = new FileReader("test.txt");
        }
        catch (IOException e) {
            System.out.println("Erreur d'ouverture du fichier");
            System.out.println(e.getMessage());
            System.exit(0);
        }
        char T[]=new char[1000];
        int n;
        while ((n=f.read(T))> 0) {
            String S=new String(T,0,n);
            System.out.print(S);
        }
    }
}
```

Fonctions d'écriture : classe **FileWriter**

Classe utilisée :

FileWriter

Constructeur :

FileWriter ([String](#) NomDeFichier)

```
→ FileWriter f = new FileWriter("test.dat");
```

→ écrasement du contenu du fichier si existant

→ les erreurs d'ouverture peuvent aussi être détectées
par l'intermédiaire de la clause : try ... catch

Méthodes :

void write (int c)	Ecrit un caractère dans le fichier.
void write (char[] Tchar)	Ecrit un tableau de caractères
void write (char[] Tchar, int Debut, int Nombre)	Ecrit une portion d'un tableau de caractères
void write (String str)	Ecrit une chaîne de caractères.
void write (String str, int Debut, int Nombre)	Ecrit une portion d'une chaîne de caractères.

Exemple :

```
import java.io.*;

class ES03 {
    public static void main(String args[]) throws IOException
    {
        FileWriter f;
        f = new FileWriter("test.dat");
        f.write("ceci est une chaine ");
        f.write(65);
        f.write('a');
        f.write(" 30.58");

        float y = (float)45.65;
        String S=Float.toString(y);
        f.write(S); //ou
        f.write("\" + y);
        //ou encore f.write("\" + y);
        f.close();
    }
}
```

Autres fonctions de la classe **FileReader**

- 1- [skip](#)(long n); Saut de n caractères. n doit être un nombre positif;
- 2- [close](#)(); Fermeture du fichier

Autres fonctions de la classe **FileWriter**

- 1- [flush](#)(); vider le tampon d'écriture dans le fichier
- 2- [close](#)(); Fermeture du fichier

Ouverture en mode Mise à jour : classe

RandomAccessFile

Pour l'ajout, la modification, accès direct etc... on utilise la classe « **RandomAccessFile** ». Celle-ci peut cependant être utilisée pour gérer aussi bien des fichiers textes que des fichiers binaires. Nous donnons alors dans ce qui suit un exemple d'ouverture de fichier en mode ajout. La classe sera étudiée dans le paragraphe suivant.

```
import java.io.*;

class ES04 {
    public static void main(String args[]) throws IOException
    {
        RandomAccessFile f;
        f = new RandomAccessFile("test.dat", "rw");
        f.seek(f.length());
        f.writeBytes("\r\nceci est une nouvelle ligne");
        f.write(20);
        f.writeBytes("#");
        int x = 65;
        f.write(x);
        f.writeBytes(" 30.58");
        f.close();
    }
}
```

6.3 Fichiers Binaires

Classe utilisée :

RandomAccessFile

Classe Mère :

[java.lang.Object](#)

`java.io.RandomAccessFile`

Constructeur :

RandomAccessFile (String NomDeFichier, String Mode)

avec Mode une chaîne de caractères qui précise le mode d'ouverture. Elle peut prendre deux valeurs possibles :

"r"	Ouverture en mode lecture. Si le fichier n'existe pas une exception est déclenchée.
"rw"	ouverture en mode mise à jour : Lecture+Ecriture+Ajout+Accès Direct. Si le fichier existe son contenu n'est pas écrasé et le pointeur de fichier est placé à la fin du fichier. Sinon, il est alors créé.

Exemple :

```
→ RandomAccessFile f = new RandomAccessFile("test.dat", "rw");
```

1.1 Méthodes d'accès

4.1.1 Accès direct.

void seek (long pos)	Positionner le pointeur de fichier sur l'octet N° pos.
int skipBytes (int n)	Sauter sur n octets.
long getFilePointer ()	Retourner la position en cours du pointeur de fichier.

4.1.2 Longueur du fichier.

long length ()	Retourne la longueur du fichier.
void setLength (long L)	Permet d'attribuer une nouvelle longueur L au fichier.

Remarques :

- 1- L'instruction `f.setLength(L)` permet de tronquer un fichier pour qu'il ne contiennent plus que L octets si son contenu initial était supérieur à L.
- 2- L'instruction `f.setLength(0)` permet d'écraser le contenu du fichier. Ainsi l'ouverture d'un fichier en écriture avec écrasement du contenu peut être réalisée de la manière suivante :

```
RandomAccessFile f = new RandomAccessFile(NomFichier, "rw");  
f.setLength(0) ;
```

Fermeture du fichier.

La fermeture du fichier est réalisée par l'intermédiaire de la méthode [close\(\)](#) .

Méthodes d'écriture

1	void <u>write</u> (int b)	Ecrit un caractère dans le fichier.
2	void <u>write</u> (byte[] Tbyte)	Ecrit un tableau d'octets
3	void <u>write</u> (byte[] Tbyte, int Debut, int nb)	Ecrit une portion d'un tableau d'octets
4	void <u>writeBoolean</u> (boolean v)	Ecrit un boolean codé sur 1 octet.
5	void <u>writeByte</u> (int v)	Ecrit un octet ou un caractère sous forme d'un octet (comme write).
6	void <u>writeBytes</u> (String s)	Ecrit une chaîne de caractères sous forme d'une séquences d'octets.
7	void <u>writeChar</u> (int v)	Ecrit un caractère sous forme de 2 octets (le plus fort ensuite le moins fort).
8	void <u>writeChars</u> (String s)	Ecrit une chaînes de caractères codé chacun sur 2 octets.
9	void <u>writeDouble</u> (double v)	Ecrit un double sous son format binaire 8 octets.
10	void <u>writeFloat</u> (float v)	Ecrit un réel sous son format binaire 4 octets.
11	void <u>writeInt</u> (int v)	Ecrit un entier sous son format binaire 4 octets.
12	void <u>writeLong</u> (long v)	Ecrit un entier long sous son format binaire 8 octets.
13	void <u>writeShort</u> (int v)	Ecrit un entier court sous son format binaire 2 octets.
14	void <u>writeUTF</u> (String str)	Ecrit une chaîne sous format d'encodage UTF-8.

Exemple :

Dans cet exemple, nous écrivons la même information (30.58) délimitée par les deux symboles < et > avec différents formats :

```
RandomAccessFile f;  
  
f = new RandomAccessFile("test.dat", "rw");  
f.writeBytes("<");  
f.writeFloat((float) 30.58);  
f.writeBytes(">\r\n<");  
f.writeDouble(30.58);  
f.writeBytes(">\r\n<");  
f.writeBytes("30.58");  
f.writeBytes(">\r\n<");  
f.writeChars("30.58");  
f.writeBytes(">\r\n<");  
f.writeUTF("30.58");  
f.writeBytes(">");
```

➔ Le fichier créé aura le contenu suivant :

<Aô£ï>	écrit avec writeFloat (4 octets)
<@%"éGë£~>	avec writeDouble (8 octets)
<30.58>	avec writeBytes (format texte : 1 octet par caractère)
< 3 0 . 5 8>	avec writeChars (format texte : 2 octets par caractère)
< ï30.58>	avec writeUTF (texte UTF : avec entête de 2 octets)

Méthodes de lecture

1	int read ().	Lit un octet. Retourne -1 si fin de fichier.
2	int read (byte[] cbuf)	Lit un tableau d'octets et retourne le nombre de caractères lus
3	int read (byte[] Tchar, int debut, int Nombre)	lit une séquence d'octets dans une portion de tableau et retourne le nombre de caractères lus.
4	boolean readBoolean ()	Lit un boolean (1 octet).
5	byte readByte ()	Lit un octet

6	char <u>readChar()</u>	Lit un caractère Unicode (2 octets)
7	double <u>readDouble()</u>	Lit un double (8 octets)
8	float <u>readFloat()</u>	Lit un réel (4 octets)
9	void <u>readFully()</u> (byte[] b)	Lit exactement b.length octets. Si la fin de fichier est détectée avant, une erreur (IOException) est déclenchée.
10	void <u>readFully()</u> (byte[] b, int off, int len)	Lit exactement len octets. Si la fin de fichier est détectée avant, une erreur (IOException) est déclenchée.
11	int <u>readInt()</u>	Lit un entier (4 octets)
12	<u>String readLine()</u>	Lit une ligne entière à partir d'un fichier texte
13	long <u>readLong()</u>	Lit un entier long (8 octets)
14	short <u>readShort()</u>	Lit un entier (2 octets)
15	int <u>readUnsignedByte()</u>	Lit un octet non signé (valeur positive)
16	int <u>readUnsignedShort()</u>	Lit un entier non signé (2 octets de valeur positive)
17	<u>String readUTF()</u>	Lit une chaîne enregistrée sous le format standard UTF-8

Exemple 1 : Lecture d'un fichier texte par tampon à l'aide d'une seule instruction.

```

RandomAccessFile f;

f = new RandomAccessFile("test.txt", "r");

int L =(int) f.length\(\); //suppose que la taille du fichier est
<32535

byte T[] = new byte[L];
    int n = f.read(T);
String S=new String(T);
System.out.println(S);
f.close();

```


Exemple 2 : Lecture d'un fichier texte ligne par ligne.

```
RandomAccessFile f;  
f = new RandomAccessFile("test.txt", "r");  
String Ligne;  
int n=0;  
while ((Ligne=f.readLine()) != null)  
    System.out.println(Ligne);  
f.close();
```

6.4 Fichiers Binaires d'enregistrements - Persistance des objets -

Classes utilisées :

ObjectInputStream : pour la lecture

ObjectOutputStream : pour l'écriture

Classe Mère :

[java.lang.Object](#)

java.io.InputStream

java.io.ObjectInputStream

[java.lang.Object](#)

java.io.OutputStream

java.io.ObjectOutputStream

Constructeur :

```
ObjectInputStream (new FileInputStream(NomDeFichier))
```

```
ObjectOutputStream (new FileOutputStream(NomDeFichier))
```

6.5 Lecture/Ecriture d'un objet

L'objet à lire ou à écrire doit appartenir à un objet **sérialisable**. La classe de l'objet doit alors implémenter l'interface **Serializable**.

Exemple :

```
class Produit implements Serializable {
    public String ref;
    public String desig;
    public double PU;
    public int QS;
    Record() {}
    Record(String ref,String deisig,double PU,int QS) {
        this.ref=ref;
        this.desig=desig;
        this.PU=PU;
        this.QS=QS;
    }
}
```

Ecriture : // Sérialisation

```
Produit p =new Produit( ...) ;
```

```
FileOutputStream fos = new FileOnputStream(NomDeFichier);
```

```
ObjectOutputStream fout = new ObjectOutputStream (fos);
```

```
fout.writeObject (p) ;
```

Lecture : //Désérialisation

```
FileInputStream fis = new FileInputStream(NomDeFichier);
```

```
ObjectInputStream fin = new ObjectInputStream (fis);
```

```
Produit r = (Produit)fin.readObject ();
```