Introduction au langage Java

Le langage Java est un langage généraliste de programmation synthétisant les principaux langages existants lors de sa création en 1995 par *Sun Microsystems*. Il permet une programmation orientée-objet (à l'instar de SmallTalk et, dans une moindre mesure, C++), modulaire (langage ADA) et reprend une syntaxe très proche de celle du langage C.

Outre son orientation objet, le langage Java a l'avantage d'être **modulaire** (on peut écrire des portions de code génériques, c-à-d utilisables par plusieurs applications), **rigoureux** (la plupart des erreurs se produisent à la compilation et non à l'exécution) et **portable** (un même programme compilé peut s'exécuter sur différents environnements). En contre-partie, les applications Java ont le défaut d'être plus lentes à l'exécution que des applications programmées en C par exemple.

I. Environnement de développement JAVA

1. 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 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 qu'elle machine ou plate-forme. Il suffit que celle-ci dispose d'un programme de traduction adéquat. Celui-ci constitue en fait ce que l'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 palte-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.

Java est un langage interprété, ce qui signifie qu'un programme compilé n'est pas directement exécutable par le système d'exploitation mais il doit être interprété par un autre programme, qu'on appelle interpréteur. La figure 1 illustre ce fonctionnement.

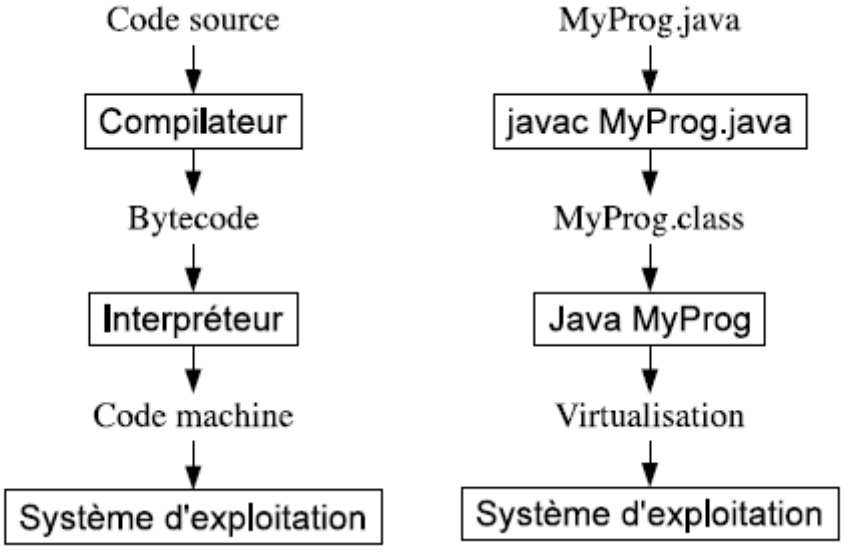

Figure 1 – Interprétation du langage

Un programmeur Java écrit son code source, sous la forme de classes, dans des fichiers dont l'extension est .java. Ce code source est alors compilé par le compilateur javac en un langage appelé *bytecode* et enregistre le résultat dans un fichier dont l'extension est .class. Le bytecode ainsi obtenu n'est pas directement utilisable. Il doit être

interprété par la *machine virtuelle* de Java qui transforme alors le code compilé en code machine compréhensible par le système d'exploitation.

C'est la raison pour laquelle Java est un langage portable : le bytecode reste le même quelque soit l'environnement d'exécution.

Pour plus de détails, nous présentons dans le schéma suivant le déroulement de développement et d'exécution d'un programme Java :

1- On écrit un programme java sauvegardé dans un fichier d'extension .java (exemple Test01.java : supposé contenir une classe nommée Test01)
2- On compile le programme pour générer le bytecode associé :
c:\ … > javac Test01.java
Un fichier Test01.class est créé.
3- Exécution du programme résultant :
c:\ … > java Test01.class

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 un fichier .class associé à chacune des classes du programme est non au programme lui-même. Si ainsi le programme Test01.java contient de classes 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.class

En 2009, Sun Microsystems est racheté par Oracle Corporation qui fournit dorénavant les outils de développement Java SE (*Standard Edition*) contenus dans le *Java Development Kit* (JDK).

2. le JDK (Java Developpement Kit)

Le JDK est l'ensemble des programmes nécessaires pour le développement d'applications Java. Il regroupe ainsi les programmes javac.exe, java.exe, appletviewer pour exécuter les applets, ainsi que d'autres classes et utilitaires de développements.

Remarque :

Nous remarquons que Sun a voulu changer la nomination du JDK en la remplaçant à partir de sa version 1.2 par SDK (Software Developpemnt Kit) ou plus précisément J2SDK pour « Java ™ 2SDK ». Avec Java 2 désignant le JDK1.x ($x \geq 2$). En fait Sun distingues entre deux concepts :

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

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

Les versions de JDK actuellement sont :

- 1996 : JDK 1.0
- 1997 : JDK 1.1
- 1998 : JDK 1.2, appelé Java 2
- 2000 : JDK 1.3
- 2002 : JDK 1.4
- 2004 : JDK 1.5, appelé Java 5
- 2006 : JDK 1.6, appelé Java 6
- 2011 : JDK 1.7, appelé Java 7
- 2014 : JDK 1.8, appelé Java 8.
- Septemnbre 2017 : JDK 9.

- mars 2018 : JDK 10.
- septembre 2018 : JDK 11.
- mars 2019 : JDK 12.
- Etc.

3. Le JRE (Java Runtime Environment)

Le JRE est 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 nécessaire 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.

4. 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 S***tring*** 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 que le symbole « . » est utilisé à la place de symbole anti-slash « \ » pour désigner les sous packages.

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

import java.util.* ;

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

import java.util.Vector ;

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

Remarques :

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

ii. 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***.

II. Eléments de base du langage Java

1. Conventions d'écriture

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 :

i. Les noms de packages sont entièrement en minuscule. Exemples :

java.awt
javax.swing
javax.swing.filechooser

ii. 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

iii. 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

iv. Une propriété est en principe un membre privée 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.

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

Exemples :

PI
MIN_Value

vi. Les primitives (types de base) et les mots sont entièrement en minuscule :

Byte, int, …
while, for, if
this, super, try, catch, length, …
class, extends, implements, …

2. Différences avec le langage C++

Le langage Java a éliminé plusieurs structures de langage C dont on peut citer :

i. Le mot clé const remplacé par final
ii. Le mot clé goto
iii. La notion de variables globales
iv. Les unions
v. les instructions : #include et #define
vi. Les opérateurs : * et &
vii. La désallocation
viii. Les paramètres par défaut
ix. Liste d'arguments variables

3. Structure générale d'un programme Java

Le squelette général d'un programme Java se présente comme suit :

```
class NomDeClasse1 {
     // Variables d'instance de la classe
     // Méthodes de la classe
}
class NomDeClasse2 {
     // Variables d'instance de la classe
     // Méthodes de la classe
}
. . .
class NomDeClassePrincipale {
     // Variables d'instance de la classe
     // Méthodes de la classe
     Public static void main (String Args[ ] ) {
           // code de programme principale
     }
```

```
}
```

Cependant, il est à remarquer que le compilateur Java génère un fichier de classe (.class) pour chacune des classes qui constitue le programme. Il est donc d'usage d'écrire chaque classe séparément (une classe par fichier). 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 dans le même **package**). Dans le cas contraire (différents répertoires ou packages), on utilisant l'instruction **import** pour préciser le chemin des classes accessibles.

4. Modificateurs et visibilités

Nous remarquons que la méthode main() qui constitue le point d'entrée de toute application est précédé 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 associés à une donnée ou une fonction membre : modificateurs de synchronisation, de visibilité, de permanence, de constance et d'abstraction.

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

Exemples :

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

i. 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 même objet réalisé en exclusion mutuelle.

ii. 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'exterieur de la classe où elle est définie.
- Une donnée ou méthode **protected** est protégé de tout accès externe comme private sauf à partir des classes dérivées. Une méthode ou donnée protected est donc accessible dans une classe fille.

Remarques :

Lorsqu'on redéfinie une méthode dans une classe fille, il est obligatoire de conserver son modificateur de visibilité ou d'utiliser un privilège d'accès plus fort c'est-à-dire :

- Une méthode **public** reste **public**,
- Une méthode **protected** reste **protected** ou devient **public,**
- Une méthode **private** teste **private** ou devient **protected** ou **public.**

iii. 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 une 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 …) 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 une 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 **variables 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 dons pas accéder à une donnée non-static ou une autre méthode non-static** :

Exemple :

```
class Classe1 {
      static int x=50 ;
      int y= 70 ;
      static void p1 (){
            x=x+10 ; // correcte
            y=y+20 ; // incorrecte
            p2 () ; // incorrecte
      }
      Void p2 () {
            x+=5 ; // correcte
            y-=10 ; // correcte
            p1 () ; // correcte
      }
}
```

iv. 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 Constante {
      public static final int CONST1=50 ;
      public static final double PI=3.14 ;
}
```

Remarque :

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

v. Modificateur abstract :

Ce modificateur permet de définir une méthode abstraite. C'est-à-dire 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 classeAbstraite {
          . . .
}
```

Remarques :

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 se fait on ne peut pas donner d'existence à une méthode abstraite déclarée private. Ce qui innterdit la combinaison private- abstract.

5. La méthode main

La remarque que nous avons fait à propos des méthodes static reste valable pour la méthode 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. Cedernier n'est pas une méthode static ou 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  Classe1 {
        static int x= 50
        int y=10 ;
        Classe1(){
            x=x+20 ;   // correcte
            y=y+30 ;   // correcte
            p2 () ;    // correcte
        }
        void p2 () {
            System.out.println ("x= "+ x+ "      y  = "+ y) ;
        }
        Public static void main (String args [ ] ) {
           new Classe1() ;
            y=y+10 ;  // incorrecte
            p2 () ;   // incorrecte
```

```
        }
}
```

III. Syntaxe du langage Java

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

- Affectation : =
- Opérateurs arithmétiques : +, -, *, /, %, ++, --, +=, -=, *=, /=, %=, . . .
- Opérateurs de relation : <, <=, >, >=, !=, ==
- Opérateurs sur le bit : &, |, ~, ^, <, >>, >>>
- Instructions de choix :
  if (condition) traitement 1 ; else traitement 2 ;
  switch { . . . } {case . . . default}
- Opérateurs logiques : &&, ||, !
- Les boucles :
  +for {. . . } {. . .}
  +Boucle for each (introduite par le JDK 5.0)
  A partir de JDK 5.0 java dispose d'une autre boucle nommée for... each. Elle ne peut être utilisée qu'au parcours des éléments d'une collection ou d'un tableau. Cette boucle s'utilise en lecture seule, ainsi, on ne pas l'utiliser pour modifier un tableau ou une collection
  +while (condition) { . . . }
  +do { . . . } while (condition)
- Transtypage :
  variable=(type) Expression ;

Le langage C a servi de base pour la syntaxe du langage Java :

- Le caractère de fin d'une instruction est " ;" exemple : a = c + c ;
- Les commentaires (non traités par le compilateur) se situent entre les symboles "/*" et "*/" ou commencent par le symbole "//" en se terminant à la fin de la ligne

```
int a ; // ce commentaire tient sur une ligne
int b ;
ou
/*Ce commentaire nécessite
2 lignes*/
int a ;
```

- les identificateurs de variables ou de méthodes acceptent les caractères {a..z}, {A..Z}, $, _ ainsi que les caractères {0..9} s'ils ne sont pas le premier caractère de l'identificateur. Il faut évidemment que l'identificateur ne soit pas un mot réservé du langage (comme int ou for).

Exemple : mon_entier et ok4all sont des identificateurs valides mais mon-entier et 4all ne sont pas valides pour des identificateurs.

- Entrées-Sorties :
  +*Affichage* : System.out.print(...) , System.out.println(...)
  +*Lecture* :

Lecture d'un entier :

```
import java.util.Scanner ;
Scanner keyb = new Scanner(System.in) ;
int i = keyb.nextInt() ;
```

Lecture d'une chaîne :

```
Scanner keyb = new Scanner(System.in) ;
String str = sc.nextLine();
```

De façon générale, pour récupérer un type de variable, il suffit d'appeler next<Type de variable commençant par une majuscule> sauf le type char.

Exemple: nextDouble, nextByte, ….

IV. Héritage en Java

I. La classe Object

L'héritage est concept qui est toujours utilisé en Java. Ainsi à chaque fois que l'on définisse 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 chaine de caractère.

Voici la liste des méthodes de la classe Object :

```
public  class Object {
     public Object() {...}  // contructeur

     public String toString() {...}

     protected  native Object clone()  throws CloneNotSupportedException {...}

     public equals(java.lang.Object) {...}
     public  native  int hashCode() {...}

     protected  void finalize()  throws Throwable {...}

     public  final  native Class getClass() {...}

     // méthodes utilisées dans la gestion des threads
     public  final  native  void notify() {...}
     public  final  native  void notifyAll() {...}

     public  final  void wait(long)  throws InterruptedException {...}
     public  final  void wait(long,  int)  throws InterruptedException {...}
}
```

-*La méthode **clone()***: Le rôle de la méthode clone() est de dupliquer un objet rapidement, en dupliquant la zone mémoire dans laquelle il se trouve à l'aide d'un processus rapide. Pour cloner un objet, il suffit donc d'appeler cette méthode, qui nous renverra une copie conforme de cet objet.

- *La méthode **toString()***: La première de ces méthodes est la méthode toString(). Cette méthode est utilisée par la machine Java toutes les fois où elle a besoin de représenter un objet sous forme d'une chaîne de caractères. Par exemple, il est légal d'écrire la ligne suivante :

```
Personne per =  new Personne ("Madani",  "Ali",  25) ;
System.out.println("Personne : " + per) ;
```

La méthode println de l'objet System.out prend en paramètre un objet de type String. La machine Java a donc besoin de convertir ("Personne : " + per) en objet de type String. Cela passe par la conversion de l'objet *per* en objet String. Cette conversion s'effectue par appel à la méthode **toString()** de la classe Object.

Il est donc presque équivalent d'écrire le code précédent et celui-ci :

```
System.out.println("personne : " + per.toString()) ;
```

- *La méthode **equals()*** : Cette méthode permet, de comparer deux objets, et notamment de savoir s'ils sont égaux. Supposons que nous avons créé deux objets de la classe Personne, qui possédaient même nom, même prénom et même âge, et nous les comparons avec ==.

```
Personne p1 =  new Personne ("Madani",  "Ali",  25) ;
Personne p2 =  new Personne ("Madani",  "Ali",  25) ;
boolean b= (p1==p2) ;
System.out.println("résultat : " + b) ; // la valeur de b est false car == compare les
                                        //adresses mémoire des objets.
```

Notons que == compare les adresses mémoire des objets, et dans ce cas (p1==p2) renvoyait false. Ce comportement est logique, mais il serait utile d'avoir à disposition un moyen de comparer des objets qui puisse nous dire que si leurs champs sont égaux, alors ces objets sont égaux. En d'autres termes, remplacer une égalité technique en égalité sémantique. C'est l'objet de la méthode **equals()**.

*- La méthode **hashCode() :*** Le rôle de la méthode hashCode() est de calculer un code numérique pour l'objet dans lequel on se trouve. Ce code numérique est censé être représentatif de l'objet, nous allons expliciter ce point immédiatement. Techniquement, la méthode hashCode() est une méthode native qui permet de calculer un nombre (int) unique associé à une instance de n'importe quelle classe. Par défaut, la méthode de la classe Object retourne l'adresse à laquelle est rangé cet objet, nombre effectivement unique, puisqu'on ne peut pas ranger deux objets au même endroit en mémoire. En toute rigueur, ce point dépend de la JVM que l'on utilise, mais c'est le cas dans la JVM de Sun.

Le point délicat est le contrat qui lie les méthodes equals() et hashCode() dans les spécifications de Java.

- Deux objets égaux au sens de equals() doivent retourner le même hashCode() ;
- Il n'est pas nécessaire que deux objets différents au sens de equals() retournent deux hashCode() différents...

Donc, surcharger la méthode equals() d'une classe entraîne systématiquement la surcharge de la méthode hashCode(). Ne pas respecter cette règle revient à s'exposer à des bugs obscurs et très difficiles à corriger

- *La méthode **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.

- *La méthode **getClass()*** : Cette méthode retourne un objet, instance d'une classe particulière appelée Class. Tout est objet en Java, y compris les classes elles-mêmes ! Il existe donc une classe Class, qui modélise les classes Java.

Notons que la méthode toString() de la classe Class est surchargée, mais ne retourne pas le nom de la classe, comme on pourrait s'y attendre.

```
Personne per =  new Marin("Madani",  "Ali") ;
System.out.println("Classe de per : " + m.getClass()) ;

> Classe de per :  class Personne
```

Si l'on veut juste le nom de la classe, il faut invoquer sa méthode getName().

```
Personne per =  new Marin("Madani",  "Ali") ;
System.out.println("Classe de per : " + m.getName()) ;

> Classe de per : Personne
```

- *Les méthodes **wait** , **notify** et **notifyAll*** : servent à coordonner des threads. Les méthodes wait (il y en a trois) mettent en attente le thread en cours d'exécution et les méthodes notify et notifyAll servent à interrompre cette attente.

Remarque :
Puisque toutes les classes Java (existantes ou crées 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 qu'il 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("abcd")) ;
L.add(new Integer(20)) ;
```

2. Héritage

Pour réaliser une nouvelle classe B qui hérite d'une classe A, la syntaxe utilisée est la suivante :

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

Un objet de la classe B peut maintenant appeler les méthodes publiques de B et de A. Si une méthode est définie à la fois dans B et dans A (redéfinition), alors celle invoquée par un objet de la classe B est la méthode redéfinie dans la classe B.

3. Mot clé this

Le mot clé this désigne l'objet en cours et peut être utilisé dans les méthodes de la classe.

4. Mot clé super

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

i. Pour appeler un constructeur de la classe mère : il est alors utilisé comme **première** instruction de **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()) ;
        }
}
```

ii. Pour appeler une méthode de la classe mère, si celle-ci est définie dans la classe fille :

Syntaxe :

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

Exemple :

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

```
class B extends A{
        void p ( ){
              System.out.println("Methode 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
```

V. Classes Abstraites et interfaces

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

i. Les classes (déjà vues)

ii. **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 nomDeMéthode( paramètres) ;
        . . .
}
```

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**.

iii. Les interfaces qui sont définies par l'intermédiaire du mot clé **interface** au lieu de **class** constituant un cas particulier des classes abstraites : d'autre 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) ;
       type1 methode1 (paramètres) ;
       . . .
}
```

Implémentation d'une interface :

```
class NomDeClass implements NomInterface{
   . . .
}
```

Remarques :

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

VI. Déclaration des variables

En Java, il existe 3 catégories de types :

i. Les types de base (types primitifs)
ii. Le type Class
iii. Le type tableau

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

1. Types de base

En plus du type void qui est utilisé uniquement pour définir les procédures, le langage Java offre 8 types de base appelés aussi primitives ou types primitifs :

Type	Classe éq.	Valeurs	Portée	Défaut
boolean	Boolean	true ou false	N/A	false
byte	Byte	entier signé	{-128..128}	0
char	Character	caractère	{/uoooo../uFFFF}	/uoooo
short	Short	entier signé	{-32768..32767}	0
int	Integer	entier signé	{-2147483648..2147483647}	0
long	Long	entier signé	$\{-2^{31}..2^{31}-1\}$	0
float	Float	réel signé	$\{-3, 4028234^{38}..3, 4028234^{38}\}$ $\{-1, 40239846^{-45}..1, 40239846^{-45}\}$	0.0
double	Double	réel signé	$\{-1, 797693134^{308}..1, 797693134^{308}\}$ $\{-4, 94065645^{-324}..4, 94065645^{-324}\}$	0.0

Exemple :

```
int x=200 ;
byte b=64 ;
```

2. Classes et objets

i. ***Création :***

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

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

Ou encore :

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

Exemple :

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

ii. ***Copie :***

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

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

Exemple :

String S1= new String ("ceci est une chaine de caractères") ;
String S2=S1 ;
Les chaines de caractères constituent un cas particulier, elles peuvent en plus être initialisées à des constantes chaînes :
S2= "abcd" ;

iii. ***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 entrainera la libération de la mémoire si celle-ci n'est pas référencée par un autre objet.

3. Les tableaux

i. ***Déclaration :***

Un tableau est un objet dynamique déclaré toujours dynamiquement :

```
Type nomDuTableau[ ] ;
```

Exemple :

```
int T[ ] ;
String TS[ ] ;
```

Il s'agit d'un tableau à plusieurs dimensions on utilise la syntaxe suivante :

```
Type nomDeTableau[ ][ ]…. ;
```

Exemple : int M[ ][ ] ;

ii. ***Création :***

Les tableaux sont, comme les instances de classes, créés par l'intermédiaire de l'opérateur new tout en précisant la taille désirée :

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

Exemple : int T[ ] =new int [50] ;

iii. ***Création d'un tableau d'objets :***

Dans le cas d'un tableau d'objets, la création du tableau ne signifie pas celle des éléments du tableau. Ces derniers doivent ceux aussi créés explicitement après la création du tableau.

Exemple :

```
String TS[ ]= new String[3] ;
TS[0]=new String ("abcd") ;
```

```
TS[1]=new String ("fg") ;
TS[2]=new String ("ijklm") ;
```

iv. *Taille d'un tableau :*

Après la création d'un tableau. Sa taille ne peut plus être modifiée, sauf si on crée un nouveau tableau et on le référence par la même variable. Dans tel cas les éléments de l'ancien tableau sont perdus.

Remarque :
La taille d'un tableau peut être déterminé automatiquement par l'intermédiaire de la propriété prédéfinie pour tous les tableaux : **length**.
Exemple : int taille= T.length ;

v. *Initialisation* :

Un tableau peut aussi être créé par l'initialisation. Celle-ci se fait de la manière qu'en C : les éléments du tableau sont fournis entre les accolades.
Exemples :

```
//Création d'un tableau :
int T[]={5,7,45,87,89} ;
//Création d'une matrice à deux lignes :
int T[][]={{1,2,3},{4,5,6}}
//Création d'un tableau de deux chaînes de caractères :
String T[]={new String("abcd"), new String("klmnop")} ;
```

VII. Quelques propriétés du langage Java :

1. les tableaux tels qu'ils sont définis en Java (objets dynamiques) ne peuvent âs 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 les classes listes existantes : **LinkedList, Vector** ou **stack** du package **java.util**.
   -Manipulation de Vector
   - void addElement(Object):ajoute un élément à la fin du vecteur
   - void clear():efface tous les éléments du vecteur
   - Object elementAt(int):élément situé à une position donnée
   - int indexOf(Object):indice d'un élément donné
   - Object remove(int):efface un élément situé à une position donnée
   - void setElementAt(Object, int):remplace un élément par un objet
   - int size():retourne la taille du vecteur

3. Un tableau de char n'est pas une chaîne de caractères en Java. Il ne paut donc pas géré comme une chaîne.
   Exemple : l'écriture
   char s[]= "abc" ; est incorrecte.
4. Il est possible de convertir un tableau char[] en une chaine (String) par l'intermédiaire de l'un des constructeurs de la classe **String**.
   Exemple :
   char S1[]={'a', 'b', 'c'} ;
   String S2=new String (S1) ;

Les différentes méthodes de la classe String

En Java, les chaînes de caractères sont gérées à l'aide de la classe String fournie en standard dans le paquetage java.lang. Cette gestion diffère du langage C où les chaînes de caractères sont mémorisées comme une suite de caractères terminée par 0. En vertu du principe d'encapsulation, l'accès direct aux données d'un objet de la classe String n'est pas possible et les structures de données utilisées sont inconnues.

La classe String met à la disposition de l'utilisateur un large éventail de méthodes gérant les chaînes de caractères. Les principales méthodes sont résumées ci-dessous. Elles sont toutes publiques sinon on ne pourrait pas les utiliser.

Remarque : la chaîne de caractères d'un objet de type String **ne peut pas être modifié** . En cas de modification, les méthodes fournissent un **nouvel objet** à partir de la chaîne de l'objet courant et répondant aux caractéristiques de la méthode. Une même chaîne de caractères (String) peut être référencée plusieurs fois puisqu'on est sûr qu'elle ne peut pas être modifiée. La classe **StringBuffer** par contre permet la création et la modification d'une chaîne de caractères.

Les premières méthodes sont des constructeurs d'un objet de type String.

- **String** () : crée une chaîne vide
- **String** (String) : crée une chaîne à partir d'une autre chaîne
- **String** (StringBuffer) : crée une chaîne à partir d'une autre chaîne de type StringBuffer

Remarque : String s = "Monsieur" ; est accepté par le compilateur et est équivalent à String s = new String ("Monsieur") ;

Les méthodes ci-après permettent d'accéder à un caractère de la chaîne, de comparer deux chaînes, de concaténer deux chaînes, de fournir la longueur d'une chaîne ou de fournir la chaîne équivalente en majuscules ou en minuscules.

- **char charAt** (int n) : fournit le n ième caractère de la chaîne de l'objet courant.
- **int compareTo** (String s) : compare la chaîne de l'objet et la chaîne s ; fournit 0 si =, <0 si inférieur, > 0 sinon.
- **String concat** (String s) : concatène la chaîne de l'objet et la chaîne s (crée un nouvel objet).
- **boolean equals** (Object s) : compare la chaîne de l'objet et la chaîne s.
- **boolean equalsIgnoreCase** (String s) : compare la chaîne de l'objet et la chaîne s (sans tenir compte de la casse).
- **int length** () : longueur de la chaîne.
- **String toLowerCase** () : fournit une nouvelle chaîne (nouvel objet) convertie en minuscules.
- **String toUpperCase** () : fournit une nouvelle chaîne (nouvel objet) convertie en majuscules.

Les méthodes indexOf() fournissent l'indice dans la chaîne de l'objet courant d'un caractère ou d'une sous-chaîne en partant du premier caractère (par défaut) ou d'un indice donné. Les méthodes lastIndex() fournissent l'indice du dernier caractère ou de la dernière sous-chaîne recherchés dans la chaîne de l'objet référencé.

- **int indexOf** (int c) : indice du caractère c dans la chaîne ; -1 s'il n'existe pas.
- **int indexOf** (int, int) : indice d'un caractère de la chaîne en partant d'un indice donné.
- **int indexOf** (String s) : indice de la sous-chaîne s.
- **int indexOf** (String, int) : indice de la sous-chaîne en partant d'un indice.
- **int lastIndexOf** (int c) : indice du dernier caractère c.
- **int lastIndexOf** (int, int) : indice du dernier caractère en partant de la fin et d'un indice.
- **int lastIndexOf** (String) : indice de la dernière sous-chaîne.
- **int lastIndexOf** (String, int) : indice de la dernière sous-chaîne en partant d'un indice.

Les méthodes suivantes permettent de remplacer un caractère par un autre (en créant un nouvel objet), d'indiquer si la chaîne commence ou finit par une sous-chaîne donnée, de fournir une sous-chaîne en indiquant les indices de début et de fin, ou de créer une chaîne en enlevant les espaces de début et de fin de chaîne.

- **String replace** (char c1, char c2) : crée une nouvelle chaîne en remplaçant le caractère c1 par c2.
- **boolean startsWith** (String s) : teste si la chaîne de l'objet commence par s.
- **boolean startsWith** (String s, int n) : teste si la chaîne de l'objet commence par s en partant de l'indice n.
- **boolean endsWith** (String s) : teste si la chaîne de l'objet se termine par s.
- **String substring** (int d) : fournit la sous-chaîne de l'objet commençant à l'indice d.
- **String substring** (int d, int f) : fournit la sous-chaîne de l'objet entre les indices d (inclus) et f (exclu).
- **String trim** () : enlève les espaces en début et fin de chaîne.

Ces méthodes **static** fournissent (indépendamment de tout objet) une chaîne de caractères correspondant au paramètre (de type boolean, int, float, etc.)

- **static String valueOf** (**boolean b**) : fournit true ou false suivant le booléen b.
- **static String valueOf** (**char**) : fournit la chaîne correspondant au caractère.
- **static String valueOf** (**char**[]) : fournit la chaîne correspondant au tableau de caractères.
- **static String valueOf** (**char[], int, int**) : fournit la chaîne correspondant au tableau de caractères entre deux indices.
- **static String valueOf** (**double**) : fournit un double sous forme de chaîne.
- **static String valueOf** (**floa**) : fournit un float sous forme de chaîne.
- **static String valueOf** (**int**) : fournit un int sous forme de chaîne.
- **static String valueOf** (**long**) : fournit un long sous forme de chaîne.

  D'autres méthodes sont disponibles pour par exemple passer d'une chaîne de caractères à un tableau de byte ou de char, pour comparer des sous-chaînes, etc. De même, d'autres constructeurs existent pour créer un objet String à partir d'un tableau de byte ou de char.

5. Les types **wrappers** tel 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(x) ;
        L.add(x) ; //incorrecte
        L.add(wx) ; //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 dérniers necessite 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 au paramètres de types base, le passage est par valeur. Afin de réaliser un passage par adresse pour ces derniers on peut les envelopper dans les objets de classe que l'on crée pour ce 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.prpriété.

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 Classe1{
      private static int x ;
      private y ;
      public static void main(String[]args){
             x=20 ;  //correcte
             y=50 ; //incorrecte
             . . .
       }
}
```

Exemple 2 :

```
class Classe1{
     private x, y ;
     voit initData(){
            x=20 ;
            y=50 ;
     }
     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 Classe1{
     private x, y ;
     Classe1(){
              initData() ;
      }
     voit initData(){
              x=20 ;
              y=50 ;
     }
     public static void main(String[]args){
               new Classe1() ;
       }
}
```

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

Exemple :

```
class Classe1{
      Int x=20, y=50 ;
       . . .
}
```

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

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 d'une 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 est :"+
 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 par l'intermédiaire de la méthode static **gc()** de la classe **System**.

VIII. Gestion des exceptions en Java:

La gestion des exceptions ou erreurs 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 executé.
      // que l'exception ait été déclenché dans le bloc try ou non.
}
```

1-le bloc **try** doit être utilisé pour entourer tout code susceptible de déclencher une exception ayant nesoin 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 tr.vaers 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.