INITIATION AU LANGAGE JAVA

I. Présentation

1.1 Historique :

Au début des années 90, Sun travaillait sur un projet visant à concevoir des logiciels simples et performants exécutés dans des PDA (Personnal Digital Assistant). Le langage C++ a été retenu pour la programmation de ces logiciels, mais le langage C s'est avéré peu performant , notamment en ce qui concerne la sécurité et la facilité d'utilisation. On a donc opté pour un nouveau langage spécifique inspiré du C : le langage Java.

1.2 Atouts de Java :

- **orienté objet** : Java est un langage full object c'est-à-dire qu'il respecte une approche orientée objet de la programmation, sans qu'il ne soit possible de programmer autrement. En clair, contrairement au C++, on ne peut faire que de la programmation orientée objet avec Java
- **portable** : un programme écrit en Java sur une plate-forme peut être exécuté sans aucune modification sur un autre système, à condition bien sûr qu'un environnement d'exécution (i.e. une machine virtuelle) soit disponible sur ce dernier.
- **interprété** : Un programme écrit en Java est exécuté par un interpréteur qui traduit en temps réel les instructions Java en instructions exécutables par le système hôte. En fait, un source écrit en langage Java n'est pas exécuté tel quel : Un source Java est transformé en un fichier qui sera interprété par une machine virtuelle. Il convient de noter que la portabilité de Java découle du fait qu'il soit interprété.
- doté d'une **API évoluée** : Java est livré en standard avec une importante API (Application Programming Interface) : threads, sockets, entrées/sorties...
- **orienté réseau** : Non seulement Java dispose de fonctions standards permettant la gestion de sockets mais il est intrinsèquement prévu pour fonctionner dans un environnement réseau de type Internet/intranet, via la création d'Applets exécutées dans un navigateur web

Java est un langage qui permet de créer des applets (applications qui s'exécutent au sein d'un navigateur : browser comme internet explorer ou netscape). On peut créer des applications Java disposant d'une interface graphique via l'awt (abstract window toolkit).

1.3 Versions de Java

Il existe plusieurs versions de Java :

- Java 1.x
- Java 2.x (depuis 2002).

1.4 L'environnement de Java :

Plusieurs environnements existent pour le développement Java :

De Sun : JDK, SDK, JRE.
De Microsoft : Visual Java.
De Borland : Jbuilder.
…Tous ces environnements, installent la machine virtuelle de Java (JVM : Java Virtual machine).

1.5 Etapes de développement :

Pour faire un programme Java , en utilisant l'outil JDK de Sun par exemple :

- on doit créer le fichier source (extension .java) , par exemple : test.java en utilisant n'importe quel éditeur de texte.
- On compile le fichier avec l'utilitaire javac : javac test.java. Si le programme ne contient aucune erreur, un fichier .class est créé. Ce fichier contient un certain nombre d'instructions, appelées bytecodes ou P-codes, qui pourront être exécutées par l'interpréteur Java (implémentation de la Machine Virtuelle Java)
- On lance l'exécution du fichier compilé avec : java test.class.

Remarque : Un fichier source contenant une application java doit nécessairement avoir comme nom le nom d'une classe existant dans le fichier.

Exemple de programme Java :
Le fichier HelloWorld.java :

```
class HelloWorld {
        public static void main (String[] args)
        {
        System.out.println("Bonjour !");
    }
}
```

1.6 Machine virtuelle de Java

La capacité à exécuter une application Java sur une plate-forme donnée est obligatoirement conditionnée par l'existence d'une implémentation de la Machine Virtuelle Java (ou JVM, Java Virtual Machine) sur cette dite plate-forme.

Nous avons vu qu'un compilateur Java générait un fichier .class contenant des bytecodes ou P-codes (P pour Program). Ces bytecodes ne peuvent pas être exécutés tel quels par le processeur de la machine sur laquelle on désire lancer un programme Java compilé. Il est nécessaire d'introduire une couche logicielle ayant pour principale fonction de traduire les bytecodes en instructions exécutables par le processeur de la machine hôte. C'est cette couche que l'on appelle la Machine Virtuelle Java (JVM). On peut donc dire que, porter Java sur une plate-forme, c'est simplement porter la machine virtuelle Java. La JVM est en fait l'implémentation d'un processeur virtuel, disposant d'un jeu d'instructions propres.

II. Eléments du langage Java

2.1 Les commentaires :

On peut insérer un commentaire de différentes façons dans un programme java :

- Le symbole // précède une ligne de commentaires.
- Les symboles /* et */ encadrent plusieurs lignes de commentaires.
- La séquence @author <nom> permettent d'insérer le nom de l'auteur du programme.
- La séquence @version <version> permettent d'insérer la version du programme.

Remarque importante : En Java , comme en C, il faut respecter la casse (distinction entre majuscule et minuscule).

2.2 Types de données :

int :type entier
long : entier long
short : entier court
float : type réel (flottant)
double : flottant double
char : type caractère (en java un caractère est codé sur 16 bits, unicode, et non sur 8)
boolean : booléen

Pour déclarer une variable, il suffit de spécifier un des types précités, suivi d'un nom de variable.

Exemples :
int UnEntier;
char UnCaractere, UnDeuxiemeCaractere;
float UnReel;

A noter que chaque ligne d'instructions se termine par un point-virgule, comme en C.
Il convient d'apporter quelques précisons relatives à l'utilisation des variables en Java :

- Toutes les variables doivent être **initialisées** explicitement avant leur utilisation. En effet, le compilateur n'attribue pas par défaut la valeur 0 aux variables non initialisées.
- Les variables globales n'existent pas en Java. Rappelons qu'une variable globale est visible dans n'importe quel module d'un programme. On ne peut donc utiliser que des variables locales à une fonction ou à un bloc (cela est du au fait que Java est un langage entièrement orienté objet).

2.3 Opérations sur les variables :

affectation : =
incrémentation : ++
décrémentation : --

2.4 Les opérateurs :

Opérateurs arithmétiques : +, -, *, /.
Opérateurs logiques : ||, &&, !
Opérateurs relationnels : <, >, <=, >=, !=

2.5 Les instructions de contrôle :

L'instruction **d'alternative** :

```
If(condition)
{
        …
}
[else   {
        …
        }
]
```

L'instruction **tantque :**

```
while(condition)
{
…
}
```

L'instruction **faire-tantque :**

```
do
{
…
}
while(condition)
```

L'instruction **pour :**

```
for(i=1 ; i<limite ; i++)
{
…
}
```

L'instruction **case :**

```
Switch(expression)
{
case valeur1 : … ; break ;
case valeur2 : … ; break ;
case valeurn : … ; break ;
default : … ; break ;
}
```

2.6 Les tableaux :

On peut déclarer un tableau de deux manières en java :

int tabetudiant[] ; dans cet exemple on n'a pas précisé les dimensions du tableau.
tabetudiant=new int[20] ; on déclare un tableau de 20 éléments de type entier.

Comme en langage C, l'origine des indices est le 0.

III. Programmation orientée Objet

3.1 Notion d'objet :

Un objet informatique est un concept qui encapsule des attributs et des méthodes. Les attributs représentent les caractéristiques de l'objet et les méthodes représentent les opérations qui s'appliquent sur ces attributs.

Objet

Attributs
méthodes

3.2 Notion de classe :

Une classe est un moule qui sert à reproduire des objets semblables. On dit aussi que l'objet est une **instance** de la classe. En java, les classes obéissent à la structure suivante :

```
class <nomClasse>
{

        //attributs
        …
        //méthodes
        …
}
```

L'exemple suivant décrit une classe Voiture dont chaque objet (voiture) possède deux attributs (couleur et vitesse) et deux méthodes *AugmenteVitesse* et *DiminueVitesse* :

```
class Voiture {
// attributs
int Couleur;
int Vitesse;

// méthodes
        void AugmenteVitesse()
        {
                if (Vitesse<5)
                Vitesse++;
        }

        void DiminueVitesse()
        {
                if (Vitesse>0)
                Vitesse--;
        }
}
```

3.3 Les méthodes :

Les méthodes en Java ont la même structure qu'en langage C. Cette structure est la suivante :

```
<type retour> <nomClasse> (liste d'arguments)
{
        …
        return resultat
}
```

Exemple : La méthode Carre suivante calcule le carré d'un nombre x passé en paramètre.

```
intCarre (int x)
{
        return x*x ;
}
```

Si la méthode ne renvoie aucun résultat précis, le type de retour sera donc **void**.

3.4 Les constructeurs :

Certaines méthodes peuvent avoir un rôle particulier : par exemple celui d'initialiser les attributs. Ces méthodes sont appelées des **constructeurs**. Exemple :

```
class Exemple
{
        //attribut
        int val ;
        //constructeur
        Exemple()
        {
        val=1 ;
        }
}
```

Le nom du constructeur a le même nom que la classe. Si on créé un objet à partir de cette classe, la valeur de la variable val sera mise à 1 automatiquement, sans qu'il ne soit nécessaire d'appeler la fonction Exemple. Un constructeur n'admet jamais aucun type de retour.

3.5 Instanciation des objets :

Un objet est créé par instanciation de se classe. Exemple :

- Voiture clio = new Voiture() ;
- Exemple expl = new Exemple(1997)

Pour manipuler les attributs et les méthodes d'un objet, il faut faire précéder l'attribut ou la méthode par le nom de l'objet suivi d'un point. Exemple :

Clio.vitesse=1 : affecter 1 à l'attribut vitesse de l'objet Clio.
Clio.DiminueVitess() : appeler la méthode DiminueVitesse de l'objet Clio.

On peut utiliser l'opérateur **this** pour désigner l'objet courant. Exemple :

this.vitesse=1 : affecter 1 à l'attribut vitesse de l'objet actuel.
this.augmentevitesse : appelle la méthode de l'objet courant.

Remarque : pour respecter la philosophie de l'approche orientée objet, il est déconseillé d'atteindre directement le contenu d'un attribut d'un objet de l'extérieur. Il faut le faire par l'intermédiaire d'une méthode de l'objet lui-même.

3.6 Données et méthodes statiques :

par défaut, lorsqu'on crée deux objets à partir d'une même classe, chacun des deux objets possède ses propres attributs. Il est cependant possible de définir en Java des variables qui sont partagées entre tous les objets d'une même classe. Pour ce faire, on utilise le mot static.

Exemple :

```
class Velo
{
```

```
        int reference ;
        static int nbrevelos=0

        Velo()
        {
        reference=0 ;
        nbrevelos++ ;
        }
}
```

La variable *nbrevelos* a été déclarée statique. Si on crée deux instances de cette classe, par exemple :

Velo velo1 = new Velo()
Velo velo2 = new Velo()

Ainsi *nbrevelos* vaudra 2 puisqu'elle a été incrémentée deux fois.

Comme pour les attributs, on peut déclarer des méthodes statiques. Exemple :

```
Static void  Decrementenbrevelo()
{
if (Nbrevelos>0)
NbreVelos-- ;
}
```

Une méthode statique a une particularité : on peut l'appeler sans qu'il ne soit nécessaire d'allouer un objet de la classe dans laquelle est définie cette fonction. Ainsi on peut invoquer cette méthode simplement de cette façon : <nom_classe>.<nom_méthode> au lieu de <nom_objet>.<nom_méthode>.

3.7 Public ou Privé :

On peut définir deux types d'accès aux membres d'un objet : **public** ou **privé**. Si un membre (attribut ou méthode) est public, il est accessible à tous les autres objets.

```
class Maclasse
{
        public int x ;  //attribut public
        private void fonction()     //méthode privée
        {
        …
        }
}
```

Remarque : la protection introduite par le mot **private** ne s'applique qu'entre des classes différentes, et non pas entre des objets d'une même classe.

3.8 L'héritage :

L'héritage est un mécanisme qui permet de créer une classe à partir d'une autre classe. La première classe est appelée classe parente ou super-classe, et la seconde classe est appelée sous-classe ou classe dérivée. Lorsqu'une classe hérite d'une autre classe, elle récupère toutes les attributs et les méthodes de sa super-classe. La sous-classe peut en revanche ajouter de nouveaux membres et aussi d'en modifier certains.

Exemple : On peut dériver la classe forme géométrique *Forme* qui représente toutes les formes géométriques en une classe *Ligne* qui décrit les lignes droites. Dans cette nouvelle classe, on redéfinit la méthode afficher pour qu'elle s'adapte au cas des lignes droites. La dérivation se fait avec le mot réservé **extends**.

```
class Ligne extends Forme
{
        int x, y;
        void Afficher()
        {
        // code d'affichage d'une ligne
        }
}
```

Remarque : On peut interdire la redéfinition d'une méthode avec le mot clé final. Exemple : final void mafonction().

3.9 Les méthodes abstraites :

Java permet de définir des méthodes sans corps ; de telles méthodes sont appelées : méthodes abstraites. Une méthode abstraite doit nécessairement être redéfinie dans une sous-classe. Exemple : abstract void mamethode().

3.10 Les interfaces :

Une interface est une classe particulière dont toutes les méthodes sont abstraites et tous les attributs sont finaux. On ne peut instancier une interface, mais on doit l'implémenter.

3.11 Les packages :

Un package est un ensemble de classes. On peut créer un package grâce au mot réservé **package**.

```
package monpackage

class classe1
{

}

class classe2
{

}
```

Si on souhaite utiliser une classe d'un package, on utilise le mot réservé import. Exemple : import unpackage.uneclasse.
Si on souhaite utiliser toues les classeses, on utilise l'instruction suivante : import unpackage.*.

L'environnement JDK de Sun est livré avec 15 packages différents, dont chacun est orienté vers un domaine donné, dont :

- Java.lang : package de base.
- Java.net : gestion de réseau.
- Java.awt : interface graphique
- Java.io : entrès-sorties.

IV. Les threads

4.1 Notion de thread :

Un *thread* est une unité d'exécution plus "petite" qu'un processus. Les threads issus d'un même processus partagent le même espace mémoire, si bien qu'ils sont plus légers donc plus rapides, chacun étant doté d'une certaine priorité. De plus, un système multiprocesseurs peut exécuter les différents threads d'un même programme simultanément, un sur chaque processeur.

4.2 Manipulation des threads :

Un thread est considéré comment étant un objet en Java. Pour utiliser des threads dans un programme, il suffit d'hériter de la classe **Thread** et de redéfinir la méthode **run()**, mais il est également possible d'implémenter l'interface **Runnable**. C'est la méthode run() qui est automatiquement appelée au moment où le thread est démarré.

Exemple : un exemple de 2 threads qui comptent de 1 à 100. Commençons donc par créer une sous-classe de la classe Thread, puis créons une classe permettant de lancer les deux threads via la méthode main() :

```
// LanceCompteurs.java
//
class ThreadCompteur extends Thread {
int no_fin;
// Constructeur
ThreadCompteur (int fin) {
no_fin = fin;
}
// On redéfinit la méthode run()
public void run () {
for (int i=1; i<=no_fin ; i++) {
System.out.println(this.getName i);
}
}
}
// Classe lançant les threads
class LanceCompteurs {
public static void main (String args[]) {
// On instancie les threads
ThreadCompteur cp1 = new ThreadCompteur (100);
ThreadCompteur cp2 = new ThreadCompteur (100);
// On démarre les deux threads
cp1.start();
cp2.start();
// On attend qu'ils aient fini de compter
while (cp1.isAlive() || cp2.isAlive) {
// On bloque le thread 100 ms
try {
                                        Thread.sleep(100);
} catch (InterruptedException e) { return; }
            }
}
}
```

Une fois compilé et exécuté, ce programme affiche à l'écran :

```
Thread-1:1
Thread-2:1
Thread-2:2
Thread-2:3
Thread-1:2
Thread-1:3
Thread-1:4
Thread-1:5
Thread-1:6
Thread-2:4
(...)
Thread-2:95
Thread-2:96
Thread-2:97
Thread-2:98
Thread-2:99
Thread-2:100
```

4.3 Les différentes méthodes de la classe thread :

- void destroy() : détruit le thread courant.
- String getName() : retourne le nom du thread.
- int getPriority() : retourne la priorité du thread
- void interrupt() : interrompt le thread.
- Static boolean interrupted() : teste si le thread a été interrompu.
- void join() ou void join(long millis) ou void(long millis, int nanos) : attendre la mort du thread, ou après un millis de ms ou, ms plus ns.
- void resume() : redémarre le thread.
- void run() : contient le code à exécuter pour le thread.
- void setPriority(int newpriority) : change la priorité du thread.
- Static void sleep(long millis) ou static void sleep(long millis, int nanos) : mettre en veille le thread pend millis ms ou ms plus ns.
- Void start() : démarre un thread.
- isAlive() : retourne vrai si le thread auquel on applique la méthode est vivant (c'est à dire à été démarré par start() et que sa méthode run() n'est pas encore terminée. Le thread vivant est donc prêt, bloqué ou en cours d'exécution

4.4 Synchronisation des threads :

En plus des méthodes décrites précédemment, il existe d'autres moyens de synchroniser les threads en Java : synchronized, wait et notify.

Mot clé *synchronized*

Les problèmes d'accès concurrents se règlent en JAVA à l'aide du mot clé *synchronized*, qui permet de déclarer qu'une méthode ou un bloc d'instructions est *critique* : un seul thread á la fois peut se trouver dans une partie synchronisée sur un objet.
Ce mécanisme est implémenté par la machine virtuelle JAVA à l'aide d'un verrou (*lock*, en fait un sémaphore). Chaque objet JAVA possède un verrou. Pour exécuter une section de code synchronisée (bloc ou méthode), il faut posséder le verrou. Si un thread commence à exécuter une section synchronisée, aucun autre thread ne pourra entrer dans une section synchonisée du même objet (même par une autre méthode) tant que le verrou n'aura pas été libéré (en quittant la partie synchronisée ou en appelant la méthode wait()).
Attention, si l'on veut synchroniser une méthode pour tous les objets de cette classe (accès à des variables de classes partagées par plusieurs threads), il faut que la méthode synchronisée soit une méthode de classe (static).

Synchronisation temporelle : wait et notify

Les méthodes wait(), notify() et notifyAll() permettent de synchroniser différents threads. Ces méthodes sont définies dans la classe Object (car elles manipule le verrou associé à un objet), mais ne doivent s'utiliser que dans des méthodes synchronized.

- wait() : le thread qui appelle cette méthode est bloqué jusqu'à ce qu'un autre thread appelle notify() ou notifyAll(). Notons que wait() libère le verrou, ce qui permet à d'autres threads d'exécuter des méthodes synchonisées du même objet.
- notify() et notifyAll() permettent de débloquer une tâche bloqué par wait(). Attention, si une tâche T1 appelle wait dans une méthode de l'objet O, seule une autre méthode du même objet pourra la débloquer; cette méthode devra être synchronisée et exécutée par une autre tâche T2.

Remarque : les sémaphores n'existent pas en tant que tels en java ; il faut les implémenter.