JAVA : Syntaxe de base

1 Quelques principes

Java est un langage interprété pré-compilé. Les fichiers sources (xxx.java) sont transformés en un langage intermédiaire (xxx.class) par un compilateur (commande **javac**) qui peut être interprété soit par un interpréteur java (commande **java**) soit par un navigateur internet.

C'est un langage à objets très proche du point de vue de la syntaxe de C++.

Il possède de riches bibliothèques standard permettant de disposer de classes d'objets d'interface, d'accès à internet etc.

L'interpréteur java gère un noyau multitâche permettant de faire fonctionner en parallèle des processus (threads) pouvant partager des variables communes ou communiquer par fichier ou réseau. Il intègre les notions d'exclusion mutuelle permettant la résolution des problèmes de synchronisation et d'accès concurrents.

Java gère les exceptions (erreurs) et permet de leur associer des actions.

Java ne possède pas d'héritage multiple mais seulement un héritage de deux types : héritage à partir d'une classe normale ou à partir d'une classe virtuelle (appelée **interface**) ne contenant que des méthodes. Il est possible d'utiliser les deux types d'héritages à la fois.

Java distingue la déclaration d'un nom d'objet de celle de l'objet lui même. Ainsi il est possible d'avoir plusieurs noms pour un même objet ou des objets n'ayant pas de nom. Les pointeurs n'existent pas de façon explicite toutefois les noms d'objets en java se comportent comme des pointeurs (en C++ par exemple). De sorte que l'on ne maîtrise que leur allocation (opération **new**) et pas leur desallocation qui est gérée par java.

2 Syntaxe de base

La syntaxe de java est très proche de celle de C++.

2.1 Commentaires

Ils sont introduits par // et se terminent avec la ligne.

On peut aussi utiliser /* en début et */ en fin ce qui permet d'étendre les commentaires sur plusieurs lignes ou de faire des commentaires sur une partie de la ligne seulement.

2.2 Types primitifs

Les variables de type primitif ne sont pas des objets et ne se comportent donc pas comme des objets mais comme des types semblables aux autres langages.

Les types primitifs sont : **char int byte short long float** et **double** comme en C mais on trouve aussi le type **boolean** qui prend les valeurs true et false.

Java offre aussi des classes correspondant à ces **types (Character**, **Integer**, **Byte**, **Short**, **Long**, **Float**, **Double** et **Boolean**) . Elles permettent d'avoir un comportement standard de tous les objets utilisés.

2.3 Tableaux

Il est possible de définir des tableaux d'objets ou de types primitifs. Le dimensionnement du tableau se fait lors de sa création par **new** et non lors de la déclaration de son nom.

Définition d'un nom de tableau : nomDeClasse nomDeTableau**[]**; ou nomDeClasse**[]** nomDeTableau;

Création du tableau : nomDeTableau = **new** nomDeClasse**[**dimension**]**;

La taille d'un tableau est désignée par nomDeTableau.**length**

Remarque : Afin de dimensionner les tableaux il faut utiliser **new** même lorsqu'il s'agit de tableaux dont les éléments sont de type primitif . Par exemple : int t[]; t=new int[25]; // définit un tableau de 25 entiers

2.4 Classes prédéfinies

Java possède une grande quantité de classes regroupées par grands thèmes dans des bibliothèques standard. On y trouve la plupart des choses dont on peut avoir besoin. Par contre il souvent assez difficile de savoir quoi et où chercher.

2.4.1 Les bibliothèques de java

les classes prédéfinies en Java sont standardisées et placées dans des bibliothèques dont les noms sont eux mêmes standard :

- **java.lang** contient les classes de base (chaînes de caractères, mathématiques, tâches, exceptions ...)
- **java.util** contient des classes comme vecteurs, piles, files, tables ...
- **java.io** contient les classes liées aux entrées/sorties texte et fichier
- **java.awt** contient les classes pour les interfaces (fenêtres, boutons, menus, graphique, événements ...)
- **javax.swing** contient les classes pour les interfaces (évolution de awt)
- **java.net** contient les classes relatives à internet (sockets, URL ...)
- **java.applet** contient les classes permettant de créer des applications contenues dans des pages en HTML

Pour pouvoir utiliser les classes de ces bibliothèques il faut y faire référence en début de fichier en utilisant la directive **import** suivie du nom de la classe de la bibliothèque ou de * pour accéder à toutes les classes de cette bibliothèque: par exemple **import java.io.***
Certaines de ces bibliothèques contiennent des sous-bibliothèques qu'il faut explicitement nommer (par exemple **java.awt.events.*** pour les événements)

Remarque : les classes de la bibliothèque de base **java.lang** n'ont pas besoin de la commande **import**.

2.4.2 Les chaînes de caractères

Elles sont définies dans les classes **String** et **StringBuffer** de **java.lang**. La première est utilisée pour les chaînes fixes et la seconde pour les chaînes variables.
On y trouve, en particulier, les méthodes suivantes

- création d'une chaîne initialisée de classe String : new String(″valeur de la chaîne″);
- création d'une chaîne initialisée à partir d'un tableau de caractères : new String(tableau);
- création d'une chaîne initialisée de classe StringBuffer : new StringBuffer(String);
- création d'une chaîne non initialisée de classe StringBuffer : new StringBuffer(taille);
- création d'une chaîne à partir d'une variable primitive : String nom = String.**ValueOf**(variable);
  ou StringBuffer nom = StringBuffer.**ValueOf**(variable);
- comparaison de chaînes : boolean **equals**(String)
- recherche d'un caractère et obtention de son rang : int **indexOf**(char)
- recherche d'un caractère et obtention de son rang en faisant démarrer la recherche à partir d'un rang donné en second paramètre : int **indexOf**(char, int)
- recherche d'une sous chaîne et obtention de son rang : int **indexOf**(String)
- recherche d'une sous chaîne et obtention de son rang en faisant démarrer la recherche à partir d'un rang donné en second paramètre : int **indexOf**(String, int)
- extraction d'un caractère par son rang : char **charAt**(int)
- extraction d'une sous chaîne par rangs de début et de fin : String **subString**(int,int)
- longueur de la chaîne : int **length**()
- concaténation : on peut utiliser l'opérateur + ou la méthode String **concat**(String)

Pour les chaînes variables on a de plus les méthodes **append** et **insert**.
Cette liste est loin d'être exhaustive.

Transformations de chaînes en éléments simple et transformations réciproques :

- Pour transformer une chaîne en type numérique primitif il est possible d'utiliser les méthodes **Integer.parseInt**, **Long.parseLong**, **Byte.parseByte** et **Short.parseShort** qui acceptent en paramètre un objet de classe String et retournent la valeur correspondante.

Ces méthodes peuvent lever une exception de type **NumberFormatException** si la conversion n'est pas possible (pour le traitement des exceptions voir 7.).

- Pour transformer une chaîne de caractères en un tableau de caractères on utilisera la méthode **toCharArray()** de la classe **String** qui retourne un tableau de caractères. L'opération inverse peut être obtenue grâce au constructeur de **String** acceptant un tableau de caractères en paramètre.

2.4.3 Le graphique

Il fait appel à la classe **Graphics** de java.awt. On y trouve les méthodes habituelles de dessin. La couleur du tracé peut être définie par la méthode **setColor(Color)** et on peut connaître sa valeur par **Color getColor()**.

1°) Dessiner : Les dessins se font grâce aux méthodes suivantes :

Ligne : **drawLine**(x1, y1, x2, y2) // coordonnées des 2 extrémités

Rectangle vide : **drawRect**(int,int,int,int) // coordonnées (x et y) du coin supérieur gauche et dimensions (largeur et hauteur)

Rectangle plein : **fillRect**(int,int,int,int) // même fonctionnement

Rectangle à bords arrondis vide : **drawRoundRect**(int,int,int,int,int,int) // coordonnées du coin supérieur gauche (x et y), dimensions (largeur et hauteur), largeur et hauteur de l'arrondi

Rectangle à bords arrondis plein : **fillRoundRect**(int,int,int,int,int,int) // même fonctionnement

Ovale ou cercle vide : **drawOval**(int,int,int,int) // coordonnées du coin supérieur gauche (x et y) et dimensions du rectangle contenant cet ovale (largeur et hauteur)

Ovale ou cercle plein : **fillOval**(int,int,int,int) // même fonctionnement

Arc d'ovale ou de cercle : **drawArc**(int,int,int,int,int,int) // coordonnées du coin supérieur gauche (x et y), dimensions du rectangle contenant cet arc d'ovale (largeur et hauteur) et angles de début et de fin de tracé en degrés

Arc d'ovale ou de cercle plein : **fillArc**(int,int,int,int,int,int) // même fonctionnement

Polygone vide : **drawPolygon**(int[], int[], int) // liste des coordonnées en x, liste des coordonnées en y des points et nombre de points

Polygone plein : **fillPolygon**(int[], int[], int) // même fonctionnement

Effacement d'une zone rectangulaire : **clearRect**(int,int,int,int) // coordonnées du coin supérieur gauche (x et y) et dimensions du rectangle à effacer (largeur et hauteur)

Ecriture de caractères : **drawString**(String, int, int) dessine le texte donné dans le 1er paramètre. Les deux derniers paramètres désignent les coordonnées (x et y) à partir desquelles sera dessiné ce texte (y est utilisé pour placer le bas du texte). La couleur et la fonte utilisées sont celles associées à l'objet **Graphics**.

2°) Les images : La classe **Graphics** permet l'affichage d'images au format gif ou jpeg. pour cela on dispose de la méthode **drawImage**(Image, int, int, ImageObserver) dont les paramètres sont les suivants : image à dessiner, coordonnées du coin supérieur gauche où la placer et composant qui gère l'image (c'est en général celui dont on a utilisé la méthode **getGraphics()**). On peut ajouter deux paramètres après les coordonnées du coin supérieur gauche pour définir les dimensions désirées de l'image (en horizontal et vertical), elle sera alors être déformée pour adopter ces dimensions.

La classe **Image** permet de faire quelques traitements élémentaires sur les images en voici les principaux :

int getHeight(ImageObserver) retourne la hauteur de l'image en pixels.

int getWidth(ImageObserver) retourne la largeur de l'image en pixels.

Image getScaledInstance(int, int, int) retourne une copie redimensionnée de l'image. les deux premiers paramètres précisent la hauteur et la largeur désirées pour la copie. Si l'un d'entre eux vaut –1 il sera adapté pour conserver l'aspect initial de l'image. Le dernier paramètre indique l'algorithme à utiliser pour créer cette copie. Il peut prendre les valeurs :

Image.SCALE_DEFAULT algorithme standard
Image.SCALE_FAST algorithme rapide
Image.SCALE_SMOOTH algorithme qui préserve la qualité de l'image
Image.SCALE_REPLICATE algorithme simple qui se contente de dupliquer ou d'enlever des pixels
Image.SCALE_AVERAGING algorithme qui utilise une méthode basée sur la moyenne entre pixels voisins.

2.4.4 Les couleurs

En java les couleurs sont des objets de la classe **Color**. Cette classe est dotée d'un constructeur permettant de créer une couleur en donnant les proportions de ses composantes rouge, verte et bleue (ces proportions sont données par des nombres compris entre 0 et 255 que l'on peut considérer comme des pourcentages):

Color maCouleur = new **Color**(120 , 255 , 22);

Réciproquement on trouve des méthodes permettant de récupérer les composantes rouge verte et bleue d'une couleur ces méthodes sont **int getRed()** , **int getGreen()** et **int getBlue()**.

Les méthodes **Color darker()** et **Color brighter()** retournent une couleur plus foncée (resp. plus claire) que l'objet dont on a invoqué ces méthodes.

Il existe d'autres méthodes liées aux couleurs qui ne seront pas détaillées ici.

2.4.5 les fontes de caractères

En java les fontes de caractères sont des objets de la classe **Font**. Cette classe est dotée d'un constructeur permettant de créer une fonte en donnant son nom, son style et sa taille. Le style est une somme des constantes **PLAIN**, **BOLD** et **ITALIC** la taille est donnée en points :

Font maFonte = new **Font**(″ Serif″ , **Font.BOLD**+**Font.ITALIC**, 14);

Les fontes traditionnellement disponibles sont Dialog, DialogInput, Serif, SansSerif et Monospaced. Les 3 dernières correspondent à Times, Helvetica et Courrier.
Les informations concernant les tailles des textes écrits peuvent être obtenues grâce à la méthode **getFontMetrics()** de la classe Graphics. Cette méthode retourne un objet de classe **FontMetrics** qui possède en particulier les méthodes suivantes :

int getHeight() qui donne la hauteur (en pixels) d'une ligne de texte
int charWidth(char ch) qui donne la longueur (en pixels) du caractère passé en paramètre
int stringWidth(String str) qui donne la longueur (en pixels) de la ligne passée en paramètre

Il existe d'autres méthodes liées aux fontes qui ne seront pas détaillées ici.

2.5 Constantes

Elles sont déclarés par le mot clé **final** par exemple : **final** int x=7;

2.6 Opérateurs

Ce sont les mêmes qu'en C++ :

= est l'opérateur d'affectation
+ - * et / sont les opérateurs arithmétiques
< <= > >= == et **!=** sont les opérateurs de comparaison. Attention la comparaison porte sur les valeurs s'il s'agit de types primitifs mais sur les noms s'il s'agit d'objets. Si l'on veut comparer le contenu d'objets il faut les doter (si ce n'est pas déjà fait) d'une méthode de comparaison généralement appelée **equals**.
! || et **&&** sont les opérateurs logiques
Enfin, on trouve des opérateurs d'incrémentation (++) et de décrémentation (**--**).

3 Classes

Tout objet doit appartenir à une classe. Il en constitue une instance. Les membres d'une classe (variables et méthodes) peuvent se voir doter d'un niveau de protection par utilisation des mots clés (**public**, **private** et **protected**). Les niveaux de protection font référence à la notion de paquetage. On peut placer certaines classes dans un paquetage en mettant, au début du fichier contenant cette classe **package** nomDuPaquetage;

Lorsqu'aucune protection n'est mise l'accès est possible depuis toute classe du même paquetage

- Avec **public** l'accès est possible depuis toute classe du même paquetage ou pas
- Avec **protected** l'accès est possible depuis toute classe du même paquetage et toute classe fille
- Avec **private** l'accès n'est possible que dans la classe elle-même

Remarque : Lorsqu'une classe n'est associée à aucun paquetage, elle appartient au paquetage par défaut de java..

3.1 Définition d'une classe

La classe est définie dans un fichier **.java** dont le nom doit être le même que celui de la classe (par exemple ici le fichier devra s'appeler **Cible.java**) :

```
class Cible;          // nom de la classe
{
      public int coordX , coordY;          // deux entiers accessibles partout
      private StringBuffer nom;            // une chaîne non accessible de l'extérieur
      public void init()                   // une méthode sans paramètres
        {
         // corps de cette méthode
        }
      public void init(int,int)            // une autre version de la méthode avec paramètres
        {
         // corps de cette méthode
        }
      void deplace(int,int)                // une autre méthode accessible
        {                                  // dans le même paquetage
         // corps de cette méthode
        }
  }
```

Plusieurs méthodes peuvent porter le même nom à condition qu'elles se distinguent par leurs paramètres (surdéfinition). Ceci permet de définir plusieurs façons de demander à l'objet de réaliser une certaine

opération. Ainsi dans l'exemple précédent on dispose de deux méthodes init l'une avec et l'autre sans paramètres.

Attention : java distingue les majuscules et les minuscules dans les noms de classes et de variables mais aussi de fichiers.

3.2 Utilisation d'objets

Pour pouvoir utiliser un objet il faut :

- Définir un nom pour cet objet : on utilise une déclaration sous la forme : NomDeClasse nomDObjet;
- Créer l'objet associé à ce nom par la commade **new** : nomDObjet = **new** NomDeClasse**(**paramètres**)**;
- On demande ensuite à cet objet de faire quelque chose en invoquant l'une de ces méthodes publiques : nomDObjet**.**nomDeMethode(paramètres);
- On peut aussi accéder aux membres publics de cet objet par leur nom : nomDObjet**.**nomDeMembre

```
Cible a;            // déclaration d'un nom (a) devant être associé à des objets de classe Cible
a=new Cible();      // création d'une instance d'objet de classe Cible associée au nom a
a.init(4,5);        // appel de la méthode init de a avec paramètres
Cible b;            // déclaration d'un nom (b) devant être associé à des objets de classe Cible
b=a;                // a et b sont deux noms pour le même objet
b.init();           // appel de la méthode init de b (c'est à dire de a) sans paramètres
b=new Cible();      /* création d'une instance d'objet de classe Cible associée au nom b qui n'est
                       dès lors plus un autre nom pour a mais bien un nouvel objet */
if (b.coordX==5) // accès à un élément de b (coordX) qui est un entier, possible car il est public
```

3.3 Nom de l'instance courante

Lorsque l'on est dans une méthode d'un objet le nom **this** est toujours un nom de cet objet.

3.4 Appartenance à une classe

On peut tester l'appartenance d'un objet à une classe par l'opérateur **instanceof** de la façon suivante :

```
If (tresnaHori instanceof Cible) { /* tresnaHori est une cible */ }
else { /* tresnaHori n'est pas une cible */ }
```

3.5 Classe interne

On peut déclarer une classe à l'intérieur d'une autre comme suit :

```
class Exterieure {
    public class Interieure {
            // contenu de la classe Interieure
            }
    // reste du contenu de la classe Exterieure
  }
```

A l'extérieur de la classe *Exterieure*, on désigne alors cette classe par son nom complet : *Exterieure.Interieure*. Bien entendu ce nom n'est accessible de l'extérieur que si la classe interne a été désignée par **public.** Si elle a été désignée **private,** elle n'est visible que dans la classe *Exterieure* () et, si rien n'a été précisé, dans le reste de l'application.

Remarque : A chaque création d'un objet x de classe *Exterieure* il y a création d'une classe *Interieure* dont on pourra par la suite créer autant d'instances d'objets que l'on voudra. Toutes seront liées à cet objet x.

4 Construction d'objets

Le constructeur est une méthode appelée lors de la création de l'objet (utilisation de **new**). Il porte le même nom que la classe et peut accepter des paramètres.

```
public Cible(paramètres) { // le constructeur de la classe Cible
        //corps du constructeur
}
```

Lorsque un objet est créé (par utilisation de **new**), le constructeur est appelé et doit recevoir ses paramètres (s'il en a). Il faut faire : **Cible a = new Cible (liste de paramètres);**

5 L'Héritage

L'Héritage permet de définir de nouvelles classes à partir de classes existantes. La nouvelle classe ainsi définie hérite des membres et des méthodes de sa classe mère et y ajoute ses propres membres et méthodes.

Un objet d'une classe B héritière de la classe A est aussi un objet de classe A et peut remplacer à tout instant un objet de classe A. En effet, un objet de classe B est au moins capable de se comporter comme un objet de classe A (il peut en général faire même plus).

Certaines méthodes de A peuvent être redéfinies dans B (on dit surchargées). Il conviendra que ces surcharges conservent la sémantique des méthodes initiales de façon à ce que le comportement d'un objet de classe B utilisé en lieu et place d'un objet de classe A reste cohérent. Si une méthode de A a été qualifiée par **final** elle ne peut pas être redéfinie dans B.

Lorsque l'on invoque une méthode d'un objet, java en recherche la définition de cette méthode dans la classe de cet objet, s'il ne la trouve pas il la recherche dans sa classe mère et ainsi de suite jusqu'à la rencontrer.

5.1 Définition d'une classe dérivée

```
class Nouvelle extends Ancienne  { // définition d'une classe Nouvelle par héritage de la classe Ancienne
        // contenu de la nouvelle classe
}
```

La conception de *Nouvelle* peut utiliser tout ce qui n'est pas privé (**private**) dans *Ancienne*. Elle n'autorise pas les accès aux membres ou méthodes privées de *Ancienne*. L'utilisateur d'un objet de la classe *Nouvelle* pourra utiliser les méthodes publiques de *Ancienne* et celles de *Nouvelle*.

5.2 Surcharge de méthodes héritées

Si des méthodes de la classe *Nouvelle* (obtenue par héritage de *Ancienne*) ont le même nom que certaines existant dans *Ancienne*, elles viennent les cacher (surcharge). Pour appeler une méthode de *Ancienne* dans *nouvelle* il faut la préfixer par **super**. Par exemple si *Ancienne* contient une méthode void calcul(int) qui est redéfinie dans *Nouvelle,* on peut appeler dans *Nouvelle* l'ancienne version de calcul par **super.**calcul(x)

5.3 Construction

Le constructeur de la classe *Nouvelle* peut faire appel à celui de la classe *Ancienne* par : **super**(paramètres). Mais ceci doit être impérativement fait au tout début du corps du constructeur de *Nouvelle* (1ere instruction).

Si ce n'est pas fait c'est le constructeur sans paramètres de la classe *Ancienne* qui sera utilisé (s'il y en a un).

Un objet d'une classe dérivée est objet de la classe de base et pas le contraire

5.4 Polymorphisme, classe Object et coercition

Un objet de classe B dérivée de A peut être utilisé partout où un objet de A peut l'être. Si une méthode de A a été redéfinie dans B on utilisera celle de B.

A quoi sert le polymorphisme ?

Si on définit une classe L qui est une liste d'objets de classe A, on pourra mettre dans cette liste des objets de n'importe quelle classe à partir du moment où elle est hérité de A.

Java introduit la classe **Object** qui est celle dont héritent implicitement toutes les classes définies en java (mais pas les types primitifs). Ainsi, tout objet peut être vu comme appartenant à la classe **Object** et donc, par exemple, être mis dans une liste dont les éléments sont de cette classe. On peut ensuite indiquer la classe réelle de cet objet en utilisant l'opérateur de coercition (nom de la classe réelle entre parenthèses) :

(nomDeClasse**)** objetDeClasseObject

qui permet de préciser que objetDeClasseObject est de classe nomDeClasse.

5.5 Les collections

Java propose une certain nombre de classes permettant de gérer des collections d'objets quelconques. Ces classes utilisent le polymorphisme et le fait que toute classe dans Java hérite de la classe **Object**. Ainsi il est possible de définir des tableaux ou des liste d'Objets dans lesquels on pourra mettre tout objet défini en Java.

5.5.1 La classe Vector

Elle permet de gérer des tableaux d'objets. Ces tableaux ont une taille qui croit au fur et à mesure que l'on les remplit. Le taux de croissance peut être défini de façon a éviter de trop fréquentes réorganisations des données en mémoire. On y trouve les méthodes suivantes :

Vector(int,int) permet de créer un vecteur en en précisant la taille initiale et le taux de croissance.
void insertElementAt(Object, int) qui ajoute un objet au rang précisé.
void addElement(Object) qui ajoute un objet à la fin du tableau.
void removeAllElements() qui vide la liste
boolean contains(Object) qui retourne true si l'objet est dans la liste
void copyInto(Object[]) qui copie le vecteur dans un tableau d'objets.
Object elementAt(int) qui retourne l'objet de rang donné (il faudra utiliser la coercition pour rétablir cet objet dans sa classe d'origine)
Object firstElement() qui retourne le premier objet (il faudra utiliser la coercition pour rétablir cet objet dans sa classe d'origine)
Object lastElement() qui retourne le dernier objet (il faudra utiliser la coercition pour rétablir cet objet dans sa classe d'origine)
int indexOf(Object, int) qui retourne le rang de la première occurrence de l'objet en commençant à partir de l'index précisé en second paramètre. Si le second paramètre est omis, le recherche commence au début.
int lastIndexOf(Object, int) qui retourne le rang de la dernière occurrence de l'objet en ne dépassant pas le rang précisé en second paramètre. Si le second paramètre est omis, le recherche va jusqu'à la fin.
void removeElementAt(int) qui supprime l'objet de rang donné.
void removeElement(Object) qui supprime la première occurrence de l'objet donné.
void removeRange(int, int) qui supprime tous les objets entre les 2 rangs donnés.
void setElementAt(int, Object) qui remplace l'objet de rang indiqué par celui passé en second paramètre.
boolean isEmpty() qui indique si la liste est vide.
int size() qui donne la taille de la liste.

5.5.2 La classe LinkedList

Elle permet de gérer des liste d'objets On y trouve les méthodes suivantes :

void add(int, Object) qui ajoute un objet au rang précisé. Si le rang est omis, l'objet est rajouté à la fin.
void clear() qui vide la liste
boolean contains(Object) qui retourne true si l'objet est dans la liste
Object get(int) qui retourne l'objet de rang donné (il faudra utiliser la coercition pour rétablir cet objet dans sa classe d'origine)
Object getFirst() qui retourne le premier objet (il faudra utiliser la coercition pour rétablir cet objet dans sa classe d'origine)
Object getLast() qui retourne le dernier objet (il faudra utiliser la coercition pour rétablir cet objet dans sa classe d'origine)
int indexOf(Object) qui retourne le rang de la première occurrence de l'objet
int lastIndexOf(Object) qui retourne le rang de la dernière occurrence de l'objet
Object remove(int) qui supprime l'objet de rang donné. La valeur de retour est cet objet.
Object removeFirst() qui supprime le premier objet. La valeur de retour est cet objet.
Object removeLast() qui supprime le dernier objet. La valeur de retour est cet objet.
Object set(int, Object) qui remplace l'objet de rang indiqué par celui passé en second paramètre. La valeur de retour est l'objet remplacé.
int size() qui donne la taille de la liste.

Remarque : Il existe bien d'autres collections dans la bibliothèque **java.utils** (piles, arbres …).

5.6 Héritage multiple

Java ne possède pas d'héritage multiple. Il existe toutefois la notion d'**interface** qui permet de doter une classe d'un certain nombre de méthodes définies dans une **interface**. De plus, on peut faire hériter une classe d'une autre et lui adjoindre une **interface**.

5.6.1 Définition d'interface

On utilise le mot clé **interface** au lieu du mot clé **class**.

```
interface NomDInterface {
    // contenu de l'interface
}
```

Une **interface** est une classe abstraite sans données qui ne peut servir qu'à être adjointe à d'autres classes.

5.6.2 Héritage et interface

Pour définir une classe qui possède une **interface** on utilise le mot clé **implements** au lieu du mot clé **extends** :

```
class Nomc implements NomDInterface {
    // contenu de la classe Nomc
}
```

On peut cumuler ceci avec un héritage normal :

```
class Nomhc extends NomDeClasseMere implements NomDInterface {
    // contenu de la classe Nomhc
}
```

6 Entrées / Sorties

6.1 Les flots

Les entrées/sorties en java se font par des flots. Ceux-ci s'appliquent aussi bien aux entrées/sorties classiques clavier et écran qu'à celles sur fichier ou sur internet.

6.1.1 Les flots standard

Les classes de base pour définir les flots sont **InputStream** et **OutputStream**. Elles possèdent une grande quantité de classes dérivées dont les plus utilisées sont les classes **PrintStream** pour les sorties et **DataInputStream** et **BufferedReader** pour les entrées.

La classe **OutputStream** n'offre que la méthode **write** permettant d'écrire un ou plusieurs octets. La classe **PrintWriter** offre en plus les méthodes **print** et **println** qui connaissent les types primitifs. Il est possible de construire un objet de la classe **PrintWriter** à partir d'un objet de la classe **OutputStream** en faisant :

PrintWriter ecriture = new **PrintWriter**(objet de classe **OutputStream**);

La classe **InputStream** n'offre que la méthode **read** permettant de lire un ou plusieurs octets. Toutefois il est possible de construire un objet de la classe **DataInputStream** à partir d'un objet de la classe **InputStream** en faisant :

DataInputStream lecture = new **DataInputStream**(objet de classe **InputStream**);

La classe **DataInputStream** offre alors les méthodes adaptées aux types primitifs comme **readInt** , **readChar**, **readLong** etc.

Pour ce qui est des chaînes de caractères, elles peuvent être transmises sans problème aux méthodes **print** et **println**. Par contre leur lecture doit faire appel à la méthode **readLine** qui n'existe pas dans la classe **DataInputStream**. Il faut alors utiliser un objet de la classe **BufferedReader** que l'on peut construire à partir d'un objet de la classe **InputStream** en faisant :

BufferedReader lectChaine = new **BufferedReader**(new **InputStreamReader**(objet de classe **InputStream**));

Remarque : toutes les méthodes permettant la lecture sur un flot peuvent lever une exception de classe **IOException** en cas d'erreur et **EOFException** s'il n'y a pas assez d'entrées pour terminer la lecture (pour le traitement des exceptions voir 7.).

6.1.2 Objets et Flots

On peut utiliser des flots pour envoyer et recevoir des objets via des fichiers ou le réseau. Pour cela on utilise les classes **ObjectOutputStream** et **ObjectInputStream.** La création de tels objets à partir d'un flot existant associé à une URL (voir 6.3), une socket ou un fichier (voir 6.4) se fait par le constructeur :

ObjectOutputStream(OutputStream)
ObjectInputStream(InputStream)

Le transfert des objets fait alors appel aux méthodes :

Objet readObject() pour la classe **ObjectInputStream.** Cette méthode peut lever les exceptions : **ClassNotFoundException**, **OptionalDataException** et **IOException**

Et **void writeObject(Objet)** pour la classe **ObjectOutputStream.** Cette méthode peut lever l'exception **IOException**

Les deux classes possèdent une méthode **close()** qui ferme le flot.

La classe **ObjectOutputStream** possède en outre une méthode **flush()** qui assure que toutes les données soient transmises.

ATTENTION : pour qu'un objet puisse être transmis par un flot il doit implémenter l'interface **Serializable** de java.io.

6.2 Clavier et écran

On utilise la classe **System** qui possède deux flux de sortie et un d'entrée. Celui d'entrée correspond au clavier ceux de sortie sont respectivement associés à l'écran et au terminal d'erreur. On les désigne comme suit :

- **System.out** est associé au terminal texte. Il est de classe **PrintStream** qui est très semblable à **PrintWriter**
- **System.err** est, en général, associé au terminal texte mais peut être redirigé. Il est de classe **PrintStream**
- **System.in** est associé au clavier. Il est de classe **InputStream** et ne permet de lire que des chaînes de caractères.

Remarque : Les navigateurs possèdent en général une fenêtre associée à ces terminaux : c'est la fenêtre 'Console Java'.

6.3 Les URL(Uniform Resource Locator)

Une **URL** (Uniform Resource Locator) permet de désigner une information sur le réseau. Elle contient le protocole (http : //), la désignation de la machine hôte (iutbay.univ-pau.fr/) parfois accompagnée d'un numéro de port (iutbay.univ-pau.fr :64111/), celle de l'information sur cette machine (java/index.html) et parfois d'une étiquette dans cette information (#etiq).
Remarque : Lorsque l'on désire désigner un fichier local on doit créer une **URL** avec le protocole file, cela s'écrit de la façon suivante : "file:///D:/java/documents/nom.txt"

Ses principales méthodes sont :

URL(String) construction à partir du texte passé en paramètre. Si ce texte ne fournit pas une URL correcte, une exception de classe **MalformedURLException** est déclenchée.

URL(URL,String) construction à partir d'une URL complétée par le texte passé en paramètre. Si ceci ne fournit pas une URL correcte, une exception de classe **MalformedURLException** est déclenchée.

URL(String, String, String) construction à partir du protocole (1er paramètre), de la machine hôte (2ème paramètre) et de la désignation de l'information (3ème paramètre). Si ces paramètres ne fournissent pas une URL correcte, une exception de classe **MalformedURLException** est déclenchée (voir 7).

String toString() qui retourne l'URL sous forme de chaîne de caractères.

String getFile() qui retourne la désignation de l'information contenue dans cette URL.

String getHost() qui retourne la désignation de la machine hôte contenue dans cette URL.

int getPort() qui retourne le numéro de port contenu dans cette URL s'il y en a un et –1 dans la cas contraire.

String getProtocol() qui retourne la désignation du protocole contenue dans cette URL.

InputStream openStream() qui établit la connexion à cette URL et retourne un flot permettant de récupérer l'information qu'elle contient. En cas d'impossibilité une exception de classe **IOException** est levée (voir 7).

URLConnection openConnection() qui crée un objet de connexion qui permettra d'établir la communication.

6.4 Communication sur le réseau par sockets en Java

Java permet de communiquer par sockets. Il utilise pour cela les classes **Socket** et **ServerSocket** placées dans la bibliothèque **java.net**.
Ces sockets permettent d'établir une connexion en utilisant un nom de machine hôte et un numéro de port puis de communiquer par des flux java standards (**InputStreamReader** et **OutputStreamReader** ou autres classes de flux).

6.4.1 La classe Socket

Elle est utilisée pour toute communication entre client et serveur. Ses principales méthodes sont :

- Socket(String, int) : construction avec un nom de machine et un numéro de port. Cette construction peut lever une exception de classe **IOException** en cas d'erreur de création de la socket et une exception de classe **UnknownHostException** si la connexion à l'hôte n'est pas possible.
- int getLocalPort(): retourne le numéro de port auquel cette socket est liée (localement).
- int getPort(): retourne le numéro de port auquel cette socket est connectée (à distance).
- void close() : fermeture de la socket.

- InputStream getInputStream() : retourne un flux pour lire sur la socket. Cette méthode peut lever une exception de classe **IOException** si le flux ne peut pas être créé.
- OutputStream getOutputStream() : retourne un flux pour écrire sur la socket. Cette méthode peut lever une exception de classe **IOException** si le flux ne peut pas être créé.

6.4.2 La classe ServerSocket

Elle est utilisée par le serveur comme socket d'écoute. Ses principales méthodes sont :

- ServerSocket(int,int) : construction et association au numéro de port donné en premier paramètre (si ce paramètre est 0 un numéro quelconque de port libre sera automatiquement choisi). Le second paramètre indique la taille de la file d'attente associée à cette socket d'écoute (s'il est omis le valeur 50 est prise). Cette construction peut lever une exception de classe **IOException** en cas d'erreur de création de la socket et une exception de classe **SecuriryException** si la création n'est pas autorisée.
- int getLocalPort():retourne le numéro de port auquel cette socket est liée (localement).
- void close() : fermeture de la socket.
- Socket accept() : attend une demande de connexion. Lorsqu'elle arrive, elle est acceptée et une nouvelle Socket est crée qui permet le dialogue avec le client. Cette méthode peut lever une exception de classe **IOException** en cas d'erreur lors de la connexion et une exception de classe **SecuriryException** si la connexion n'est pas autorisée.

6.4.3 Les flux associés aux sockets

Les méthodes getInputStream() et getOutputStream() retournent des flux de classe **InputStream** et **OutputStream**. On peut ensuite, à partir de ces flux, en créer d'autres selon le mode de communication que l'on désire employer (chaînes de caractères, types primitifs, objets …)

Exemple, pour une communication par chaînes de caractères :
En entrée : BufferedReader lire=new BufferedReader(new InputStreamReader(maSocket.getInputStream()));
En sortie : PrintWriter ecrire=new PrintWriter(new BufferedOutputStream(maSocket.getOutputStream()));
Ne pas oublier d'utiliser, à la fin de la communication, la méthode close des flux ainsi créés pour les fermer.

6.5 Les Fichiers

Ils sont représentés par la classe **File**. Ses principales méthodes sont :

File(String) construction avec le nom passé en paramètre.
createNewFile() création d'un nouveau fichier physique (il faut que le nom donné à la construction ne corresponde pas déjà à un fichier). Cette méthode peut lever une exception de classe **IOException**.
boolean mkdir() création d'un nouveau répertoire la valeur en retour est true si la création a été faite
boolean mkdirs() création d'un nouveau répertoire avec tous les sous répertoires en amont la valeur en retour est true si la création a été faite
boolean canRead() indique si le fichier peut être lu
boolean canWrite() indique si le fichier peut être écrit.
boolean delete() destruction du fichier la valeur en retour est true si la destruction a été faite.
deleteOnExit() destruction du fichier lors de l'arrêt de la machine virtuelle java.
boolean exists() indique si le fichier existe.
String getAbsolutePath() retourne le chemin absolu du fichier
boolean isDirectory() indique si le nom correspond à un répertoire.
boolean isFile() indique si le nom correspond à un fichier.
boolean isHidden() indique si le nom correspond à un répertoire ou fichier caché.
long length() retourne la taille du fichier.
boolean setReadOnly() met le fichier en mode lecture seule la valeur en retour est true si l'opération a été faite.
URL toURL() convertit le nom du fichier en URL. Cette méthode peut lever une exception de classe **MalformedURLException**

Remarque : la plupart de ces méthodes peuvent lever une exception de classe **SecurityException**.

Les accès aux fichiers font appel à des flots (voir 6.1) de classe **FileOutputStream** et **FileInputStream**. On les obtient en passant en paramètre à leur constructeur un l'objet de classe File.
FileoutputStream hérite de **OutputStream** tandis que **FileInputStream** hérite de **InputStream**. Ils offrent des accès à l'information en mode octet. Et possèdent une méthode **close()** .

Remarque : il est indispensable, lorsque l'on utilise des flots de classes plus évoluées comme **PrintWriter** pour accéder à des fichier, d'utiliser leur méthode **flush()** avant de fermer le fichier de façon à s'assurer que toutes les données ont été écrites.

7 Les Exceptions

Java utilise des exceptions pour traiter les événements pouvant survenir lors de certaines opérations. Lorsque l'on utilise de telles opérations il est impératif de prendre en compte la récupération et le traitement des éventuelles exceptions.

7.1 Capture et traitement d'une exception

Toute opération susceptible de lever une exception doit être placée dans un bloc **try**, le traitement de l'exception fait l'objet d'un bloc **catch** :

```
try { opération(s) susceptible(s) de lever une exception de classe ClasseException }
catch (ClasseException  nomException) { traitement de l'exception }
```

7.2 Déclenchement d'une exception

Une méthode qui doit lever une exception doit être définie comme suit :

```
typeDeRetour nomDeMethode (liste de paramètres) throws ClasseException
    { // dans le corps de cette méthode on trouvera :
    throw(nomException); // nomException est de classe ClasseException
    }
```

8 Application ou Applet

8.1 Application

Java permet l'écriture d'applications qui sont des programmes qui peuvent fonctionner en autonome (sans besoin d'être intégrés dans une page HTML). Ils doivent donc créer leur propre fenêtre par héritage de la classe **Frame** par exemple. Une application comporte une classe contenant une méthode **main** déclarée comme suit :

```
public static void main(String arguments[]) {
    // corps du programme principal
}
```

Remarque : une méthode déclarée **static** est en fait une fonction (comme en C) indépendante de toute classe. Il est possible de déclarer d'autres fonctions que **main** de type **static** toutefois elles doivent quand même être placées dans une classe. Elle seront appelées par : nomDeClasse**.**nomDeMethode(paramètres).

arguments est un tableau de chaînes de caractères correspondant aux paramètres d'appel de l'application (comme en C). On peut connaître la taille de ce tableau en utilisant **arguments.length**

8.2 Applet

Une applet est une application devant être placée dans un fichier HTML On doit donc l'insérer dans une page HTML en lui allouant de la place (comme on le fait pour une image). Elle peut ensuite être exécutée par un navigateur.

9 Les Threads

Java permet d'écrire des processus (threads) dont il gère l'exécution en parallèle. Ces processus peuvent communiquer entre eux par des objets communs ou par le réseau.

9.1 Définition d'un thread

La façon la plus simple de créer un processus est de créer une classe héritant de la classe **Thread** de java ou implémentant l'interface **Runnable** puis de l'intégrer à un processus de java. Son lancement se fera avec la méthode **start** du processus de java et il exécutera alors sa méthode **run**.

On va donc écrire :

```
class MonThread extends Thread {
    // contenu de la classe
    public MonThread(liste de paramètres) {
```

```
            // construction du thread
            }
        public void run() {
            // ce que fait le thread (éventuellement une boucle infinie)
            }
    }
```

9.2 Utilisation d'un thread

Pour lancer un processus (thread) il faut :

- le déclarer par MonThread maTache = **new** MonThread**(**paramètres du constructeur s'il y en a**);**
- l'inclure dans un processus java par **Thread** tache **= new Thread(**maTache**);**
- lancer ce processus java par tache**.start();**

A partir de ce moment le processus appelé maTache est lancé et il exécute sa méthode **run**.

Le processus se terminera lorsqu'il arrivera à la fin de sa méthode **run**. On peut attendre sa terminaison par **tache.join()** qui doit traiter l'exception d'interruption de type **InterruptedException**

Les autres méthodes de gestion sont :

- **Thread currentThread()** qui retourne le processus courant.
- **int getpriority()** qui retourne la priorité d'un processus. Plus elle est élevée plus le thread s'exécute souvent. Elle est comprise entre les valeurs contenues dans les membres MINPRIORITY et MAXPRIORITY du thread.
- **void setpriority(int)** qui permet de modifier cette priorité (uniquement en l'abaissant).
- **void sleep(int)** qui fait attendre le délai donné en paramètre (en ms). On peut ajouter un second paramètre entier qui donne une durée supplémentaire exprimée en ns.
- **void interrupt()** qui interrompt le processus par une exception de type **InterruptedException**
- **void yeld()** qui suspend temporairement le thread de façon à laisser les autres s'exécuter.

9.3 Communication avec les processus (threads)

La communication peut se faire par le réseau ou par objets partagées.

9.3.1 Partage d'objets

Il suffit de passer les objets que l'on souhaite partager avec le processus en paramètres à celui-ci lors de sa construction. Le constructeur du processus se chargera de conserver des noms pour ces objets afin de pouvoir les utiliser ensuite dans sa méthode **run**.

9.3.2 Conflits d'accès et synchronisation

Le fait d'avoir des processus qui s'exécutent en parallèle soulève bien évidemment le problème des accès concurrents à des ressources et de la synchronisation. La solution retenue par java est celle des moniteurs de Hoare c'est à dire de la possibilité d'écrire des méthodes dont java assure l'exécution en exclusion mutuelle. Ainsi lorsqu'un processus exécute une telle méthode pour accéder à une ressource il est sûr d'être le seul à le faire à ce moment. On définit une telle méthode par le mot clé **synchronized** :

```
synchronized typeDeRetour nomDeMéthode(liste de paramètres) {
    // corps de la méthode
}
```

Lors de l'exécution de cette méthode aucune autre méthode déclarée **synchronized** de l'objet auquel elle appartient ne peut être exécutée. On peut faire attendre les autres processus désireux d'accéder à cette ressource jusqu'à ce qu'elle se libère. Pour cela on écrira dans la méthode d'accès (qui doit être **synchronized**) :

```
try { wait(); }
catch (InterruptedException e) { /* ce qui est fait si le processus est interrompu lors de cette attente */ }
```

Le processus ayant exécuté ce code sera suspendu jusqu'à ce qu'on lui signale que la ressource est libre. Pour ce faire il faudra qu'un autre processus (celui qui libère la ressource) exécute l'une des deux méthodes suivantes :

- **Notify()** s'il veut ne libérer qu'un des processus en attente sur l'objet
- ou **NotifyAll()** s'il veut les libérer tous

Remarque : On peut donner en paramètre à **wait** une durée exprimée en ms au bout de laquelle le processus se réveille même si aucune notification n'a eu lieu. Ceci permet d'éviter qu'un processus ne reste bloqué trop longtemps en attente d'une ressource et permet de mettre en place des solutions de remplacement lorsqu'une ressource est trop longtemps indisponible.

SOMMAIRE