TD/TP 1

L'objectif de ce TD/TP est d'écrire quelques programmes simples pour se familiariser avec le langage Java. Le travail de compilation / exécution se fera en mode commande.

1 Installation de Java et préparation du travail

Sous ***Windows*** :

Installer Java en téléchargeant le JDK à :

https://www.oracle.com/java/technologies/downloads/

(Choisir JDK et ensuite la version binaire correspondant à votre OS)

Pour travailler en mode commande (**c:\>**): ***Démarrer>Exécuter>***taper ***cmd***.

Pour taper ses programmes utiliser un éditeur de texte simple ***blocNote*** ou ***VS Code*** pour une meilleure assistance syntaxique. Ouvrez d'abord votre répertoire dans une fenêtre ***Windows***.

Les commandes **`javac`** (pour compiler) et **`java`** (pour exécuter) se trouvent normalement dans le répertoire **`c:\jdk...\bin`**. Pour éviter de taper le chemin complet :

```
c:\jdk...\bin\javac PremierExemple.java
```

et taper simplement

```
javac PremierExemple.java
```

rajouter le répertoire **`c:\jdk...\bin`** dans la variable d'environnement ***Path***. Exemple :

Avant: `Path=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem`
Après: `Path=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;c:\jdk...\bin`

Aller dans ***panneau de configuration,*** dossier ***System*** ensuite l'onglet ***Avancé***. Modifier alors la variable d'environnement ***Path.***

2 Compilation et exécution d'un programme Java

NB. Créer un nouveau fichier d'extension **`.java`** pour chaque programme. Utiliser un éditeur de texte courant.

Premier programme : « Bonjour »

Ecrire le programme suivant dans un fichier **`PremierExemple.java`** :

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

- Compiler par : **`javac PremierExemple.java`**
- Exécuter par : **`java PremierExemple`**
- Constater la création du ficher **`PremierExemple.class`** résultat de la compilation. Commandes **dir**.
- Noter surtout que le nom du fichier d'extension **`.class`** vient du nom de la classe (première ligne).

Remarque : Le fichier source **.java** ici a le même nom que la classe contenant la fonction **main(),** i.e. l'identificateur qui suit le mot class dans le code source. Le nom du fichier **.class** généré pour cette classe est cet identificateur justement.

Exercice :

- Rajouter d'autres lignes pour imprimer "Bonjour, ça va ?"
- Rediriger le résultat d'exécution vers un fichier **`.txt`** qu'on visualisera à part (**`java votreClasse > resultat.txt`**).
- Créer un second fichier **`DeuxiemeExemple.java`**. On va imprimer : *Bonjour MonPrenom*. Où le nom *MonPrenom* est donné au lancement du programme sur la ligne de commande java.

```java
class DeuxiemeExemple {
    Run | Debug
    public static void main(String args[]){
        if (args.length != 0 )
            System.out.println("Bonjour " + args[0]);
        }
}
```

Exécuter avec : **`java DeuxiemeExemple MonPrenom`**

A noter : Le mot *MonPrenom* constitue le premier élément args[0] du tableau args[], paramètre de la fonction ***main***. Le champ ***length*** donne la taille d'un tableau en Java.

3 E/S de données élémentaires (Classe Scanner)

3.1 Lecture d'un simple entier avec la classe Java Scanner

```java
import java.util.*;    // Package Java qui contient la classe Scanner
class Saisie {
    //  Lecture d'un entier, version scanner
    public static void main(String args[]) {
        Scanner clavier = new Scanner(System.in);
        System.out.print("Donner un entier: ");
        int n = clavier.nextInt();
        System.out.println (n*2);
    }
}
```

La classe **Scanner** se trouve dans le package **java.util** . Elle permet de déclarer un objet, variable **clavier** ici, sur lequel on peut lire des données. On l'a instancié ici par :

```
new Scanner(System.in);
```

à partir du fichier standard d'entrée représenté par l'objet **System.in**

La méthode **nextInt()** permet de lire un entier. Pour lire un réel, **nextFloat()** ou **nextDouble()** . Pour les chaînes **nextLine()** pour lire une ligne ou **next()** pour une chaîne, etc. (**exercice:** vérifier ces méthodes). Il n'y a pas **nextChar()** !

Pour en savoir plus, voir (https://docs.oracle.com/javase/8/docs/api/).

Note : Avec **Scanner** on instancie normalement un objet à partir d'un fichier texte qui contient des données, e.g. **new Scanner(new File("resultat.txt"));** Il faut alors utiliser l'exception **FileNotFoundException** pour la classe **File**.

3.2 Lecture de plusieurs données

```
1   import java.util.*;    // Package Java qui contient la classe Scanner
2   class Saisie2 {
3       public static void main (String args[]) {
4           // Partie Déclaration
5           Scanner clavier = new Scanner(System.in);
6           int age;
7           String nom;
8           double taille;
9           // Partie lecture de données
10          System.out.print("Quelle est votre nom?: ");
11          nom = clavier.nextLine();
12          System.out.print("Quelle est votre age?: ");
13          age = clavier.nextInt();
14          System.out.print("Quelle est votre taille?: ");
15          taille = clavier.nextDouble();
16          // Partie sortie des résultats
17          System.out.println("Bonjour "+nom);
18          System.out.print("Vous avez "+age+" ans");
19          System.out.println(" et vous mesurez "+taille+" mètres");
20      }
21  }
```

Exercice :

- Créer, compiler et exécuter ce programme.
- Version 2 : Le même programme. Sortie des résultats avec format.
  Au lieu de **println()**, on peut utiliser **format()** qui a la syntaxe de **printf()** de C.
  Remplacer les trois dernières lignes **System.out.println()** par :

```
System.out.format("Bonjour %s %nVous avez %d ans", nom, age);
System.out.format(" et vous mesurez %f mètres %n", taille);
```

  NB. La méthode **System.out.printf()** existe aussi et est équivalente à **System.out.format()**. **%n** est le format pour "nouvelle ligne".

3.3 Lecture d'un caractère

On lit une chaîne avec **`next(),`** et on prend son premier caractère avec **`charAt(0).`**

```
1   class monChar {
2       public static void main(String args[]) {
3           char monChar;
4           Scanner clavier = new Scanner(System.in);
5           String s = clavier.next();
6           monChar = s.charAt(0);
7           System.out.println(monChar);
8       }
9   }
```

On peut condenser et écrire :

```
monChar = clavier.next().charAt(0);
```

sans déclarer explicitement la chaîne **s**.

Exercice :

Ecrire un programme qui lit trois paramètres : un nombre entier, un caractère représentant une opération (+, -, x ou /) et un second nombre entier, puis qui effectue le calcul ainsi indiqué et affiche le résultat. (Attention à éviter les divisions par 0). Utilisez l'instruction **switch**.

4 Programme avec traitement et usage de fonctions

Conversion d'une température donnée en degré Celsius, vers une température en degré Fahrenait.

a. version 1 : Sur le même modèle que le programme précédent, créer un programme Java (fichier **`Celsius.java`**) qui effectue cette conversion. Utiliser la formule :

```
f = 9./5 * c + 32
```

où f est la t° *Fahrenait* et c la t° *Celsius*.

NB. Pour que la division 9/5 s'effectue en réel, utiliser **9.** au lieu de **9** dans le code Java.

b. version 2 : Usage de fonction en Java (**static**, dans la même classe).

Dans la classe **`Celsius.java`**. Ajouter maintenant la fonction (**méthode**) :

```
static double c2f(int c){
    double f = 9./5 * c + 32;
    return f;
}
```

(à ajouter après la fonction main() avant l' } de fin de la classe Celsius) et remplacer l'instruction initiale : **`f = 9./5 * c + 32;`** par : **`f = c2f(c);`**

Compiler et exécuter. Discuter.

c. version 3 : Mettre maintenant la fonction **`c2f()`** ci-dessus dans une nouvelle classe que l'on appellera **`Celc.`**

```
class Celc {
    static double c2f(int c){
        double f = 9./5 * c + 32;
        return f;
    }
};
```

(à ajouter après l' }; de fin de la classe **Celsius** dans le même fichier source) et remplacer l'instruction initiale (programme **main** toujours) : **`f = c2f (c);`** par : **`f = Celc.c2f(c);`**

Compiler et exécuter. Discuter.

d. Version 4 : (Ne pas imiter cet exemple.) Maintenant, enlever le mot **static** dans la définition de la fonction **c2f()** de la classe **Celc :** (**double c2f(int c)** au lieu de **static double c2f (int c)**).

Compiler et exécuter. Discuter.

Remplacer maintenant l'instruction initiale (programme **main** toujours) : **`f = Celc.c2f(c);`** par : **`f = obj.c2f(c);`**

où **`obj`** est un objet à déclarer auparavant par: **`Celc obj = new Celc();`**

Compiler et exécuter. Discuter.

A Noter : Dans ce dernier cas, on doit d'abord instancier un objet **obj** de la classe **Celc** pour pouvoir appeler (donc lui appliquer) la fonction **c2f()**, dite ***méthode d'instance*** dans ce cas. Mais comme, il n'y a pas de données propres à chaque objet de cette classe **Celc**, il n'y a pas besoin d'instancier un objet pour utiliser la méthode **c2f()**. C'est pour cette raison qu'on peut la déclarer **static**. Comme cela on l'appelle *sans instancier* d'objets. On dit ***méthode de classe*** dans ce cas.

Remarque : Noter aussi qu'on pourrait, dans le premier cas, instancier plusieurs objets ***obj1, obj2, ...*** de classe ***Celc***. L'appel **objn.c2f(c)** serait indifférent de l'objet auquel il s'applique. Ce qui explique pourquoi il y a des fonctions **static** et justifie la méthode de classe dans la Version-3 ci-dessus.

5 Surcharge de fonctions

En principe, une méthode a un nom unique dans une classe. Cependant Java permet à une méthode d'avoir le même nom que d'autres grâce au mécanisme de surcharge (ang. ***overload).*** Java utilise leur signature pour distinguer entre les différentes méthodes ayant le même nom dans une classe, c'est à dire la liste des paramètres. Ce sont le nombre, le type et l'ordre des paramètres qui permettent de distinguer ces différentes méthodes.

Soit la classe Data :

```
class Data {
    static void draw(String s) {
        System.out.println("Ceci est une chaîne: "+s);
    }
    static void draw(int i) {
        System.out.println("Ceci est un entier: "+i);
    }
    static void draw(double f) {
        System.out.println("Maintenant un double: "+f);
    }
    static void draw(int i, double f) {
        System.out.format("Une entier %d et un double %f %n",i,f);
    }
}
```

Les différents appels suivants correspondent aux bonnes fonctions :

```
Data.draw ("Picasso");     // 1ère méthode, draw(String)
Data.draw (1);             // 2e méthode, draw(Int)
Data.draw (3.1459);        // 3e méthode, draw(double)
Data.draw (2, 1.68);       // 4e méthode, draw (int, double)
```

Exercice : Ecrire un programme vérifiant ces différentes fonctions.

A noter : Le paramètre retour d'une fonction ne permet pas de distinguer entre deux fonctions. La surcharge est surtout utile pour définir plusieurs constructeurs pour un objet (voir chapitre 3).

Méthodes à nombre variable de paramètres : Varargs

Un aspect particulier de la surcharge et la déclaration d'un nombre arbitraire de paramètres. Cela se fait par (trois points ...). Soit l'exemple :

```
public static void f(char c, int... p) {
    System.out.println(c + " " + p.length);
    for (int i=0; i<p.length; i++) System.out.println("  " + p[i]);
}
```

Les trois points (...) doivent apparaître après le dernier paramètre. Ici, on a un premier paramètre char et ensuite 0, 1 ou plusieurs entiers dans une variable p. **`int...`** signifie un nombre quelconque de paramètres entiers. Au fait, le **varargs** remplace favorablement un paramètre tableau dont la taille est bornée. C'est comme un tableau mais de taille illimitée (sauf par le système). D'où l'usage possible de **p.length** dans la fonction pour connaître la taille actuelle du paramètre **p**. La fonction imprime ensuite ses paramètres.

Les appels suivants sont valides

```
char c='a';
f(c);
f(c, 1);
f(c, 2,3,4);
int[] monTableau = {5,6,7,8 };
f(c, monTableau);
```

Exercice : Vérifier cet exemple et chercher d'autres cas.

6 Les Wrappers

Classes **`Integer`**, **`Float`**, **`Boolean`** etc. du package **`java.lang`**. Ce sont des sous-classes de la classe **`Number`**. Ces classes détiennent principalement des méthodes pour convertir entre elles des données numériques, surtout vers (ou à partir de) leur forme chaine de caractères.

a) On considèrera le cas de la classe **Integer**, les autres sont semblables.

- **String** vers **Integer**. Fonction **valueOf()**. Correspondance entre la chaîne "123" et l'entier Integer de valeur 123.

```
Integer I;
I = Integer.valueOf("123");
```

- **int** vers **Integer**. Correspondance inverse de **int** vers **Integer**.

```
Integer I = new Integer (i);    // Par constructeur
```

  - ❖ Discuter (The constructor Integer(int) has been deprecated since version 9).
  - ❖ Essayer maintenant de convertir l'entier **123** en un objet Integer de valeur **123** : `Integer I = 123 ; // Autoboxing`

    N.B. L'autoboxing fait référence à la conversion d'une valeur primitive en un objet de la classe wrapper correspondante.

- **String** vers **int**. Fonction **parseInt()**. Correspondance entre la chaîne "456" et l'entier de valeur 456.

```
int i;
i = Integer.parseInt("456");
```

- **Integer** vers **int**. Fonction **intValue()**. Correspondance entre objet **Integer** et entier **int**.

```
int i; Integer I;
i =  I.intValue();
```

  - ❖ Remarquer que c'est une méthode d'instance ici (fonction d'accès).
  - ❖ Essayer maintenant une affectation directe : `i = I ; // Auto-unboxing`

    N.B. L'auto-unboxing fait référence à la conversion d'un objet d'un type wrapper en sa valeur primitive correspondante.

- **Integer** ou **int** vers **String**. Passage réciproque de **Integer** ou **int** vers une chaîne **String.**

  Méthode générale **valueOf()** de la classe **String** s'appliquant (par surcharge) à tous les types primitifs.

```
int i = 34;
s = String.valueOf(i);      // s devient "34"
```

  Méthode **toString()** de la classe **Integer** ici.

```
String s;
Integer I = 345;
s = I.toString();       // s devient "345"
```

  Cette méthode est intéressante car elle est héritée de la classe **Object** et peut donc s'appliquer à tout objet si elle est redéfinie. On peut l'utiliser à profit pour imprimer un objet **`println(objet.toString);`**.

- Comparaison entre deux objets **Integer**. Résultat **int**.

```
int i = I.compareTo(J);    // 0 si I = J, négatif si I<J, positif si I>J
```

  On peut aussi utiliser les opérateurs de comparaison comme **I < J.**

b) Le cas des autres classes est analogue. Testez-les (**`Float`**, **`Double`**, …)

7 Exercice de synthèse

L'objectif de cet exercice est de programmer une classe Pile (pile de caractères), représentée par un tableau et un indice de sommet de pile.

Compléter la classe suivante (à créer dans un fichier **Pile.java**) :

```
1   public class Pile {
2       //  Déclarations des attributs de la pile
3       static final int MAX = 8;
4       char t[];
5       int top;
6       //  Programmation des méthodes de la pile
7       public Pile() {
8           // Initialise une pile vide
9           t = new char[MAX];
10          top = -1;
11      }
12      public void empiler(char c) {
13          // Empile le caractère donné en paramètre
14          if (!estPleine())
15              t[++top] = c;
16          else
17              System.out.println("Pile pleine");
18      }
19      public char sommet() {
20          // Retourne le caractère au sommet de la pile, sinon '\0' ... à completer ...
21      }
22      public void depiler() {
23          // décapite la pile (retire le sommet ) ... à completer ...
24      }
25      public boolean estVide() {
26          // Teste si la pile est vide ... à completer ...
27      }
28      public boolean estPleine() {
29          // teste si la pile est pleine ... à completer ...
30      }
31  }; // class Pile
```

1) Ecrire un programme qui lit une chaîne de caractère et l'imprime inversée.

Algorithme :

```
Pile p, char c;
lire (c);
Tant que (c != '#')
   empiler c sur p;
   lire (c)
ftq
Tantque (pile non vide)
   c = sommet de p;
   écrire (c);
   dépiler p;
ftq
```

- Créer le programme dans un fichier **`TestPile.java`**
- La fin de la chaîne à lire est marquée par le caractère **#**.

2) Ecrire un programme qui lit un texte contenant une série de parenthèses et qui
- à la rencontre du caractère **(** il l'empile sur une pile p
- à la rencontre du caractère **)** il dépile la pile p
- ignore tout autre caractère.
- s'arrête à la lecture du caractère **#**.

A la fin du texte lu, si la pile est vide l'expression est bien parenthésée. Sinon, il y a plus de parenthèses ouvrantes que de parenthèses fermantes. Si la pile est vide prématurément, lors d'un dépilement, alors il y a plus de parenthèses fermantes que de parenthèses ouvrantes.