Programmation par les Objets en Java Travaux Dirigés 1

Najib Tounsi

(Lien permanent: http://www.emi.ac.ma/ntounsi/COURS/Java/TD/tdJava1.html (.pdf))

L'objectif de ce TD est d'écrire quelques programmes simples pour se familiariser avec le langage Java. Le travail de compilation / exécution se fera en mode commande, e.g. *Terminal* de *Linux* / *MacOS*, ou *cmd* de *Windows* .

Avant propos

L'élève est censé(e) savoir comment créer un répertoire (commandes `md` ou `mkdir`, `cd`, etc.) et son équivalent graphique (créer / ouvrir un nouveau dossier etc.), lister son contenu (commande `dir` ou `ls`), savoir utiliser un éditeur de texte pour créer un programme et sauvegarder un programme source sous un répertoire donné. On doit aussi connaître les commandes de base *Unix* (*Linux* ou *MacOS*) et *Windows* (`cat`, `type`, `more` etc.)

Sommaire

Installation de Java et préparation du travail
Compilation exécution d'un programme Java
E/S de données élémentaires
Programme avec traitement et usage de fonctions
Surcharge de fonctions
Quelques classes de bibliothèque
Les Wrappers
Objets vs Valeurs
Passage des paramètres en Java
Annexe

0) Installation de Java et préparation du travail.

Sous *MacOS* ou *Linux*, Java est généralement déjà fourni. (le vérifier en tapant `javac -version`. On devrait voir `javac 1.6.0_29` ou une version supérieure). Sous *Windows*, voir l'annexe: Préparation du TP sur PC

1) Compilation exécution d'un programme Java

NB. Créer un nouveau fichier d'extension `.java` pour chaque programme. Utiliser un éditeur de texte courant. Aide à la présentation syntaxique. Par exemple, Sublime (https://www.sublimetext.com/3).

Programme *hello world*

Ecrire le programme suivant dans un fichier `HelloWorld.java` :

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

- Compiler par: `javac HelloWorld.java`
- Exécuter par: `java HelloWorld`
- Constater la création du ficher `HelloWorld.class` résultat de la compilation. Commandes *ls* ou *dir*.
- Noter surtout que le nom du fichier d'extension `.class` vient du nom de la classe (ligne 1).

Exercice: rajouter d'autres lignes pour imprimer "bonjour" en d'autres langues (e.g. "Bonjour, ça va?", "سلام").
(Sur la ligne commande Windows, taper d'abord `chcp 65001` pour changer le code page vers Unicode UTF-8.)

A Noter: Le fichier source a ici 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.

Version 2: Créer un second programme `Hello2.java`. On va imprimer: `Bonjour` *`Untel`*. où le nom *Untel* est donné au lancement du programme sur la ligne commande `java`.

```
class Hello2 {
       static public void main(String args[]){
       if (args.length !=0)
           System.out.println("Hello " + args[0]);
       }
  }
```

Exécuter avec : `java Hello2 Fatima`

A noter: Le mot *Fatima* 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.

2) E/S de données élémentaires (classe `scanner`)

2.1) Lecture d'un simple entier avec la classe Java `scanner`.

```
import java.util.*;    // Package Java qui contient la classe scanner

class Saisie  {
    /**
     *  Lecture d'un entier, version scanner
     */

    static public  void main(String args[]) {
        Scanner clavier = new Scanner(System.in);
        System.out.print("Donner 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 instancie cet objet par `new Scanner` à partir d'un fichier texte, e.g. `monFichier.dat` , par

```
new Scanner(new File("monFichier.dat"));
```

(qui doit utiliser l'exception `FileNotFoundException`)

ou à partir du fichier standard d'entrée représenté par l'objet `System.in` (qui ne nécessite pas d'exception)

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

La méthode `nextInt()` permet de lire un entier. Pour lire un réel, `nextFloat()` ou `nextDouble()` (exercice: le vérifier). Pour les chaînes `nextLine()`, etc. Voir (https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html). Il n'y a pas `nextChar()` !

2.2) Lecture de plusieurs données.

```
import java.util.*;    // Package Java qui contient la classe scanner
class Saisie2 {
    static public void main(String args[]) {
        // Partie Déclaration
        Scanner clavier = new Scanner(System.in);
        int age;
        String nom;
        double taille;

        // Partie lecture de données
        System.out.print("Quelle est votre nom?: ");
        nom = clavier.nextLine();
        System.out.print("Quelle est votre age?: ");
        age = clavier.nextInt();
        System.out.print("Quelle est votre taille?: ");
        taille = clavier.nextDouble();

        // Partie sortie des résultats
        System.out.println("Bonjour "+nom);
        System.out.print("Vous avez "+age+" ans");
        System.out.println(" et vous mesurez "+taille+" mètres");
    }
};
```

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.

Exercice: Remplacer les trois dernières lignes `System.out.print(ln)` 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".

2.3) Lecture d'un caractère.

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

```
import java.util.Scanner;
class monChar {
    static public void main(String args[]) {
        char monChar;
    Scanner clavier = new Scanner(System.in);
        String s = clavier.next();
        monChar = s.charAt(0);
        System.out.println(monChar);
    }
}
```

On peut condenser et écrire:

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

sans déclarer explicitement la chaîne *s*.

3) Programme avec traitement et usage de fonctions

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

a. Sur le même modèle que le programme précédent, créer un programme Java (fichier `Celsius.java`)

```
class Celsius {
      public static void main(String args[]){
         ...
       }
}
```

qui effectue cette conversion. Utiliser la formule:

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

où *f* est la t° *Fahrenait* et *c* la t° *Celsius*. Appeler la classe `Celsius`, et le source `Celsius.java`.

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

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

Dans le source `Celsius.java`. ajouter maintenant la fonction (on dit aussi *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 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 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: "Mauvais" exemple. Maintenant, enlever le mot `static` dans le profile de la fonction `c2f()` de la classe `Celc`
(`double c2f(int c)` au lieu de `static double c2f(int c)`)

et remplacer l'instruction initiale (programme *main* toujours)

```
f = Celc.c2f (c) ;
```

par

```
f = obj.c2f (c) ;
```

où `obj` est une variable à déclarer auparavant par:

```
Celc obj = new Celc();
```

Compiler et exécuter. Discuter.

A Noter: Dans ce dernier cas, on a été obligé d'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*. Ce qui signifie qu'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 obj_1, obj_2, ... de classe *Celc*. L'appel $obj_n.c2f(c)$ 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.

4) 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 et le type des paramètres qui permet de distinguer.

Soit la classe `DataArtist` :

```
class DataArtist {
   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 suivant correspondent aux bonnes fonctions:

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

Exercice: le vérifier.

A noter: Le paramètre retour d'une fonction ne permet pas de distinguer entre deux fonctions. `static int draw(int)` est la même signature que `static void draw(int)`. Le vérifier.

La surcharge est surtout utile pour définir plusieurs constructeurs pour un objet.

Méthodes à nombre variable de paramètres

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

```
public static void f(char c, int... p) {
    System.out.println(c + " " + p.length);
    for (int e:p) System.out.println("  " + e);
}
```

L'ellipse ... doit apparaître après le dernier paramètre. Ici, on une premier paramètre `char` et ensuite 0, 1 ou plusieurs entiers dans une variable `p`. `int ...` signifie un nombre quelconque de paramètres entiers. Au fait l'ellipse 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 (tableau donc) `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 l'exemple et chercher d'autres cas à vous.

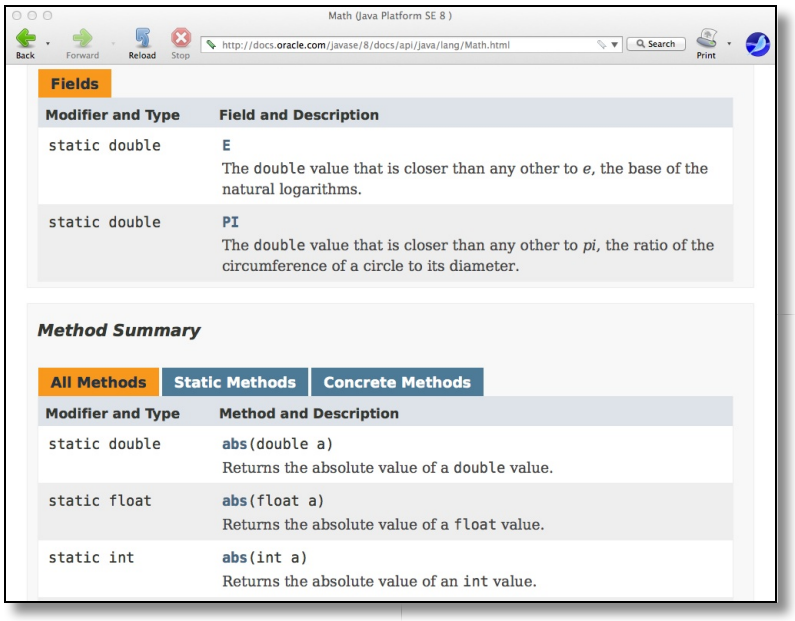
NB. C'est comme l'opérateur relationnel de *Projection*. qui admet en paramètre un nom de relation et ensuite une suite de nom d'attributs de projection.

5) Quelques classes de bibliothèque

a) La Classe **`Math`** du package **`lang`**.

```
import java.lang.Math;
```

Cette classe contient, en plus des constantes *e* et *pi*, les méthodes pour le calcul numériques (fonctions mathématiques classiques). Ce sont des méthodes toutes `static`. Exemple `double Math.abs (double)`, `double Math.sqrt (double)` etc.



(http://docs.oracle.com/javase/8/docs/api/java/lang/Math.html)

Exercice : Vérifier le programme suivant:

```
import java.lang.Math;
class TestMath {
    static public void main(String args[]) {
        System.out.println("e = " + Math.E);
        System.out.println("pi = " + Math.PI);
        int largeur = 3, longueur = 4;
       double w = Math.pow(largeur,2) + Math.pow(longueur,2);
        double hypotenuse = Math.sqrt(w);
        System.out.println("Hypoténuse = " + hypotenuse);

        //... il vaut mieux écrire largeur * largeur  que  pow (largeur,2)... Of course ;-)
    }
}
```

NB. L'instruction `import` n'est pas nécessaire ici. Les classes du package `java.lang` sont importées implicitement .

Exercice: Utiliser la fonction `static double` **`random()`**, qui retourne un nombre pseudo-aléatoire supérieur ou égal à 0.0 et inférieur à 1.0, pour imprimer 6 nombres entiers aléatoires compris entre 1 et 49 inclus.
Modifier le programme pour avoir 6 nombres tous différents (c.f. jeu du *Loto*).

b) La Classe **`Calendar`** du package **`util`**.

Cette classe abstraite contient les méthodes pour manipuler les dates dans toute ces composantes (à travers les champs `YEAR`, `MONTH`, `DAY_OF_MONTH`, `HOUR` etc.), et faire des calculs sur les dates comme déterminer le prochain jour ou semaine.

(voir http://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html pour les détails de champs et méthodes)

Exemple-1: L'objet `Calendar` et son contenu:

```
import java.util.*;
class TestCalendar {
    static public void main(String args[]) {

        Calendar rightNow = Calendar.getInstance();
             // Création d'une instance date initialisée par défaut à la date locale.

        System.out.println(rightNow);
```

```
            // le contenu de rightNow pour les curieux!
        }
}
```

Après exécution (le `Fri Mar 23 10:34:04 WET 2012`):

```
java.util.GregorianCalendar[

   … informations techniques sur le fuseau horaire,
     la zone, l'heure d'été etc.  …

firstDayOfWeek=1,
minimalDaysInFirstWeek=1,
ERA=1,
YEAR=2012,
MONTH=2,                                    <-- Mois numérotée à partir de 0
WEEK_OF_YEAR=12,
WEEK_OF_MONTH=4,
DAY_OF_MONTH=23,
DAY_OF_YEAR=83,
DAY_OF_WEEK=6,                                 <-- 6e jour de la semaine (Vendredi)
DAY_OF_WEEK_IN_MONTH=4,
AM_PM=0,
HOUR=10,
HOUR_OF_DAY=10,
MINUTE=34,
SECOND=7,
MILLISECOND=578,
ZONE_OFFSET=0,                                      <-- Décalage horaire 0, GMT.
DST_OFFSET=0
]
```

Exemple-2: Les fonctions d'accès aux champs. Les fonctions *get*(...) sont nombreuses et de la forme *get*(*quelqueChose*) où le paramètre est un champs `Calendar` (constante symbolique de type entier). Voir la documentation officielle. API ci-dessus.

```
class TestCalendar {
    static public void main(String args[]) {
        Calendar rightNow = Calendar.getInstance();

        // tester les get()
        int j = rightNow.get (Calendar.DAY_OF_MONTH);
        int m = rightNow.get (Calendar.MONTH);
        int y = rightNow.get (Calendar.YEAR);

        System.out.println("On est le:"+j+ "/" +(m+1)+"/"+y);
  }
}
```

Affiche:

```
On est le:23/3/2012
```

On a rajouté 1 pour le mois, car ils sont comptés à partir de 0. (Voir résultat exemple-1)

Exemple-3: Calcul du jour.

Dans le même programme, rajouter en les complétant les instructions suivantes pour déterminer le nom du jour de la semaine de façon à imprimer: "`On est le:`**`Vendredi`** `23/3/2012`".

```
String jour ="";
switch(rightNow.get(Calendar.DAY_OF_WEEK)){
     case 1: jour = "Dimanche"; break;
     case 2: jour = "Lundi"; break;
 // ... à compléter ...
     case 7: jour = "Samedi"; break;
 }
System.out.printf("On est le: %s %d/%d/%d%n", jour, j, m+1, y);
```

Exemple-4: Utilisation des *set*(...) pour changer les dates. Même remarque que pour les *get*(...). Voir l'API ci-dessus.

La fonction *set* (*champ* , *valeur*) permet de changer le champ (entier) d'une date

Exercice : Reprendre le même programme, et changer de date du mois vers *Mai*.

```
rightNow.set(Calendar.MONTH, Calendar.MAY);
```

et le jour du mis vers le 28.

```
rightNow.set(Calendar.DAY_OF_MONTH, 28);
```

pour imprimer la date du jour comme précédemment.

Utiliser la fonction *add* (*champ*, *valeur*) pour calculer la date du lendemain.

```
rightNow.add (Calendar.DAY_OF_MONTH, 1);
```

Exercice : Vérifier que le lendemain du 28 Février est 29 Février 2012, année bissextile, et que le lendemain du 28 Février est 1er Mars pour 2014.

Exercices: Faire d'autres exemples... Utiliser le champ `Hour_OF_DAY` .

6) Les Wrappers.

Classes `Integer`, `Float`, `Boolean` etc. du package `java.lang`. Sous-classes de `Number`.

Permettent surtout de convertir des données 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");
  ```

  Remarque: Par constructeur on peut faire la même conversion avec `new Integer ( "123"); .`

- **`String` vers `int`**. Fonction *parseInt*(). Correspondance entre `"456"` et l'entier `int` 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).

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

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

- **`Integer` ou `int` vers chaîne `String`**. Passage réciproque de `Integer` ou `int` vers 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. Par exemple:

```
float f = 12.34f;
Float F = new Float (f);      // de float vers Float
String s;
F = Float.valueOf("12.34");    // de String vers Float
f = Float.parseFloat ("12.34");   // de String vers float
f = F.floatValue();               // de Float vers float
s = F.toString();              // de Float vers String
s = String.valueOf(f);                 // de float vers String
```

...

7) Objets vs Valeurs

Les références à objets : Une affectation $x = y$ n'est pas toujours le même effet selon que ce soit un objet primitif ou non. Selon que x et y contiennent des valeurs primitives ou sont des références vers des objets.

On va le tester sur des entiers (objets primitifs) et sur un tableau (objet java).

```
//
// Objets vs Valeurs
//
class   Test {
    static public void main(String[] args) {

        int x = 1, y;

        ////////    affectation de valeurs     /////////
        y = x;   // deux valeurs égales mais objets differents.
      System.out.println("Avant (x = 100): x = " + x + " , y = " + y);
 x = 100;
 System.out.println("Après (x = 100): x = " + x + " mais y = " + y);
        // constater que y n'as pas changé

        int[] u = {4, 5};  // tableau à 2 entiers.
        int[] v;

      //////   même chose avec objets (ici tableaux)    ///////
        v = u;
        System.out.println("Avant (u[0] = 100): u[0] = " + u[0] + " ,  v[0] = " + v[0]);
```

```
    u[0] = 100;
      System.out.println("Après (u[0] = 100): u[0] = " + u[0] + " ET v[0] = " + v[0]);
        // constater que v[0] a changé aussi
    }
}
```

On obtiendra:

```
Avant (x = 100):  x = 1 , y = 1
Après (x = 100):  x = 100 mais y = 1              <-- y n'a pas changé avec x
Avant (u[0] = 100):  u[0] = 4 ,  v[0] = 4
Après (u[0] = 100):  u[0] = 100 ET v[0] = 100      <-- v a changé avec u
```

où on voit que `u` et `v` désignent le même objet: une modification de `u` est aussi une modification de `v`.

Exercice:

Selon la même idée, faire un programme qui teste l'affectation $x = y$ où x et y sont des objet d'une classe C.

Indication: Soit la classe *C*:

```
class C {
    int x;  // un champ entier
    public int getX(){return x;}       // méthode pour consulter le champ x
    public void setX(int p){x = p;}  // méthode pour modifier le champ x
}
```

Créer deux instances x et y:

```
    C  x = new C(), y = new C();
```

les initialiser et les afficher:

```
    x.setX(5);
    y.setX(6);
    System.out.println(x.getX() + " et " + y.getX());
```

Constater le résultat: `5 et 6`. Deux objets différents.

Faire maintenant:

```
    x = y;
```

ensuite modifier x et afficher x y:

```
    x.setX(4);
    System.out.println(x.getX() + " et " + y.getX());
```

Constater que y aussi a été modifié (résultat: `4 et 4`).

8) Passage des paramètres en Java

On retrouve cette même caractéristique dans le passage des paramètres en Java.

Les objets en Java sont passés en paramètre par *valeur*. Mais cette valeur peut-être une donnée (objet primitif) ou une référence à un objet.

Tester sur l'exemple suivant:

```
//
// Passage des paramètres
//

class Test extends Object {
    static public void main(String args[]){

        int x = 2;
        System.out.println("Avant modif,  x = " + x);
        modifVal(x);
        System.out.println("Apres modif,  x = " + x);

        int [] t = {2, 3};
        System.out.println("Avant modif,  t[0] = " + t[0]);
        modifObj(t);
        System.out.println("Apres modif,  t[0] = " + t[0]);
    }


    public static void modifObj(int p[]) {
        p[0] = p[0] + 200;    // Objet référencé p est modifié
    }

    public static void modifVal(int x) {
        x = x + 200;    //  paramètre x modifié
    }
}
```

On obtient:

```
Avant modif,  x = 2
Apres modif,  x = 2
Avant modif,  t[0] = 2
Apres modif,  t[0] = 202
```

NB: Quand on change le paramètre lui-même, c'est à dire la variable *p* ici, on change la référence mais pas le tableau.

Exercice 8.1: Vérifier maintenant qu'en modifiant la méthode `modifObj` pour changer `p` :

```
public static void modifObj(int p[]) {
     p = new int [] {200, 300, 400};                    // p[0] = 200
}
```

`t[0]` ne va pas changer. On aura toujours `t[0] = 2`.

Exercice 8.2 : Reprendre la classe C précédente (§7) et tester le passage de paramètre d'un objet de classe C.

Déclarer une fonction:file:///C:/Users/Najib/AppData/Local/Temp/fz3temp-1/tdJava1.html

```
    public static void modifObj(C p) {
        p.setX(p.getX() + 200);    // Objet référencé p est modifié
    }
```

et l'appeler par (en déclarant: `C o = new C();`)

```
    modifObj(o);
```

/

Evaluation

Certains exercices seront évalués à la demande et corrigés par l'encadrant du TP.

Annexe: Préparation du TP sur PC.

Installer Java en téléchargeant `Java SE Development Kit 7u3` à:

http://www.oracle.com/technetwork/java/javase/downloads/jdk-7u3-download-1501626.html

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

On pourrait souhaiter de créer un répertoire pour y sauvegarder ses programmes java. Aller vers la racine

`cd c:\`

et taper

`md votreNom`

`cd votreNom`

Pour taper ses programmes utiliser *blocNote* simple ou *wordPad*(++) ou *Sublime* pour une meilleure assistance syntaxique . Ouvrez d'abord votre répertoire dans une fenêtre *Windows*.

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

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

et taper simplement

`javac` source.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*.

Programmation par les Objets en Java
Travaux Dirigés 2

Najib Tounsi

(Lien permanent: http://www.emi.ma/ntounsi/COURS/Java/TD/tdJava2.html (.pdf))

Programmation par abstraction

Classe Pile simple.

Documentation de la classe Pile.

Ajout des cas d'exception.

Programme d'analyse d'une expression arithmétique.

1) Classe Pile simple

On va programmer la classe 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`):

```
public class Pile {
//
//  Déclarations des attributs de la pile
//
    static final int MAX = 8;
    char t[];
    int top;

//
//  Programmation des opérations (méthodes) de la pile
//
    public Pile() {
        // Initialise une pile vide
        t = new char[MAX];
        top = -1;
    }

    public void empiler(char c) {
        // Empile le caractère donné en paramètre
        if (!estPleine())
            t[++top] = c;
        else
            System.out.println("Pile pleine");
    }

    public char sommet() {
        // Retourne le caractère au sommet de la pile, sinon '\0'
        //   ... a compléter ...
    }

    public void depiler() {
```

```
        //    décapite la pile (retire le sommet )

        //   ... a compléter ...
    }

    public boolean estVide() {
        //    Teste si la pile est vide
        return (top < 0);
    }

    public boolean estPleine() {
        // teste si la pile est pleine
        //  ...   à completer ...
    }

}; // class Pile
```

Exercice-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

Indications:

- Créer le programme dans un fichier `TestPile.java`

- Utiliser

```
char monChar; Scanner clavier = new Scanner(System.in);
monChar = clavier.next().charAt(0);
```

pour lire un caractère `monChar` en entrée.

- La fin de la chaîne à lire est marquée par le caractère '#'.

Retrouver la solution ici (http://www.emi.ac.ma/ntounsi/COURS/Java/TD/TestPile.java).

Exercice-2) Ecrire un programme qui lit une 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.

2) Documentation de la classe Pile

On va créer une page Web de documentation de la classe *Pile*. Il doit y avoir

- D'abord un texte sous forme de commentaire qui documente la classe en générale, par exemple ce qu'est un objet Pile etc.
- Ensuite, il doit y avoir un texte qui documente chaque méthode en indiquant ses paramètres données, son résultat éventuel et le traitement effectué.

Dans le fichier `Pile.java` contenant la classe *Pile*, ajouter en début de fichier un commentaire qui commence par `/**`. C'est le commentaire générale de la classe. Exemple:

```
/**
 Classe Java correspondant à une pile
 Un pile est un objet sur lequel on empile ...
 etc...
 */
```

Ensuite, avant chaque déclaration de méthode, ajouter le commentaire qui documente la méthode. Exemple:

```
  /**
   * Empile le caractère donné
   */
   public void empiler(char c){...}
```

On peut alors créer les pages documentation de la classe Pile avec la commande `javadoc` :

```
javadoc Pile.java
```

Un certain nombre de fichiers sont alors créés. L'un des fichiers est `index.html` qui est la page Web principale.

On peut améliorer la documentation finale en rajoutant des lignes commentaires @param, @return, @see, pour documenter les paramètres. Exemple: TBD ...

3) Gestion des exceptions en Java

Une exception est un cas particulier (ou erreur) qui se produit dans un programme. Par exemple pile vide en cas de dépilement, rencontre d'un caractère nom numérique lors de la lecture d'un nombre etc.

L'exemple simple suivant est une méthode qui calcule la factorielle d'un entier, et qui lève ***une exception*** quand l'entier est négatif.

```
    static public int fact(int x) throws ExceptionFactNegatif {
        if ( x < 0 ) throw new ExceptionFactNegatif();
        else {
```

```
        // calcul factorielle dans int f ...
           return f;
        }
    }
```

On rajoute au profile de la méthode le mot clé **`throws`** et le type d'objet levé, qui est ici le classe `ExceptionFactNegatif`. (voir plus bas).

L'utilisation de cette méthode doit s'attendre, grâce à une instruction **`try`**, à une éventuelle levée d'exception et la capter, grâce à une instruction **`catch`**, pour pouvoir la traiter.

```
        int n = ...
        try {
            int f = fact(n);
        }
        catch (ExceptionFactNegatif e) {
            System.out.println("Valeur negative pour factorielle");
            e.printStackTrace();
        }
```

`try` indique un bloc d'instructions dans lequel une exception peut être levée "une erreur peut se produire".

Le bloc **`catch`** contient les instructions à exécuter dans le cas où un type d'exception s'est produit dans le try correspondant. Ici, on imprime une message pour la circonstance. **`catch`** est paramétré par l'exception à capter. ici objet **`e`** de classe **`ExceptionFactNegatif`**.

On peut utiliser cet objet **`e`** justement, pour suivre la trace de l'exception produite, avec `e.printStackTrace()` en plus du message à imprimer.

Le résultat de la séquence de programme précédente exécutée pour n = -1 est:

```
Valeur negative pour factorielle
ExceptionFactNegatif
        at TestException.fact(TestException.java:17)
        at TestException.main(TestException.java:5)
```

La classe correspondant à l'exception souhaitée, `ExceptionFactNegatif` ici est donnée par:

```
class ExceptionFactNegatif extends Throwable {};
```

Classe qui ne contient rien. N'est elle pas une exception?

Le fait c'est qu'elle est sous classe de la classe prédéfinie **`Throwable`**. Condition nécessaire pour pouvoir être soulevée comme exception.

A un même bloc `try` peuvent correspondre deux blocs `catch`. On peut ajouter à l'exemple ci-dessus le cas de factorielle qui dépasse la capacité d'un `int`. Factorielle 13 est déjà 6 227 020 800.

On crée donc une deuxième exception,

```
class ExceptionFactSuperieur12 extends Throwable {};
```

et on écrira:

```
        try {
```

```
            int f = fact(n);
        }
        catch (ExceptionFactNegatif e) {
            System.out.println("Valeur negative pour factorielle");
            e.printStackTrace();
        }
        catch (ExceptionFactSuperieur12 e) {
            System.out.println("Valeur > 12 pour factorielle");
            e.printStackTrace();
        }
```

Exercice-3)

Suivant ce même modèle, reprendre la classe pile et traiter les cas "Pile vide" (dans *dépiler* et *sommet*) et "Pile pleine" (dans *empiler*).

Indication:

On définira deux classes **`ExceptionPileUnderflow`** et **`ExceptionPileOverflow`** correspondant respectivement aux cas débordement de pile par en bas, pile vide, ou débordement par en haut, pile pleine.

On reprogrammera les opération de la classe pile en conséquence, par exemple:

```
    public char sommet() throws ExceptionPileUnderflow {
        // Retourne le caractère au sommet de la pile
        if (!estVide())
            return t[top];
        else
            throw new ExceptionPileUnderflow();
    }
```

etc...

On reprendra ensuite le programme `TestPile.java` avec cette fois-ci des blocs `try` / `catch`.

Une solution : PileException.java, TestPileException.java. (voir la version .html pour les liens)

Exercice-4:

a) Réfléchir à un algorithme qui vérifie si une chaîne (ou expression) est bien parenthésée: une parenthèse fermante est en correspondance avec une parenthèse ouverte. "Parenthèse" à prendre au sens large parmi les symboles: `( < [ { } ] > )`.

Exemple:

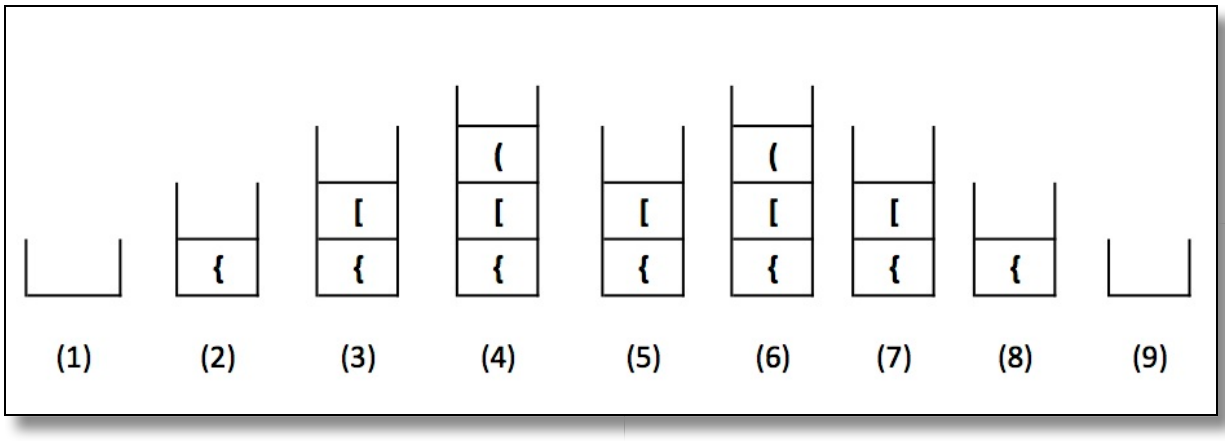
`{[(a+51)-(c)]}` est correcte

`{(a+b)]` est incorrecte (`]` ne correspond pas à `{`),
ainsi que `(a+b))` (parenthèse fermante en trop) ou
`((a+b)` (parenthèse ouvrante en trop)

Indications:

- Le programme doit ignorer (ne pas traiter) les symboles autres que les "parenthèses".

- Les étapes sont:



1. On commence par une pile vide (voir figur
2. On lit `{`, on l'empile
3. On lit `[`, on l'empile
4. On lit `(`, on l'empile
5. On ignore `a+51`, on lit `)`, elle correspond au sommet de la pile (4), on dépile celle-ci
6. On ignore `-`, on lit `(`, on l'empile
7. On ignore `c`, on lit `)`, elle correspond au sommet de la pile (6), on dépile celle-ci
8. On lit `]`, il correspond au sommet de la pile (7), on dépile celle-ci
9. On lit `}`, elle correspond au sommet de la pile (8), on dépile celle-ci. On a fini la lecture. Si on finit une pile vide, l'expression est correctement parenthésée.

b) Utiliser la pile de l'exercice précédent pour programmer cet algorithme.

That's all folks.

Evaluation

Certains exercices seront évalués à la demande et corrigés par l'encadrant du TP.

Programmation par les Objets en Java

Najib TOUNSI

Les classes en Java (TD 3)

Notions à voir: La classe et ses caractéristiques (champs/méthodes, `private`/`public`, `static`, constructeurs, documentation), les caractéristiques héritées de la classe *Object* (méthodes *equals*, *toString*).

I. Notion de classe

I.1 Classe, champs, méthodes, instanciation, `this`, `private` *VS*. `public`.

Créer une classe Point (coordonnée *x,y*) avec des méthodes *setX*(), *setY*(), *getX*(), *getY*() pour resp. affecter une coordonnée (*x* ou *y*) et accéder à sa valeur (*x* ou *y*). Mettre cette classe dans un ficher `Point.java` et le compiler.

```
/**
 * Classe Point du plan avec ses cordonnées x et y
 */

 public class Point extends Object{
    private int x, y;

    /**
     * Methode qui affecte la valeur de son paramètre
     * au Point this.
     */
    public void setX(int p) {
        x = p;
    }

    public void setY(int p) {
        y = p;
    }

    public int getX() {
```

```
        return x;
    }

    public int getY() {
        return y;
    }
};
```

1. Faire un programme test qui crée un point *p*, lui affecte des coordonnées et imprime ensuite ses coordonnées. Mettre ce programme dans un fichier `TestPoint.java` dans le même répertoire.

```
public class TestPoint{
    static public void main(String args[]){
        Point p = new Point();
        p.setX (3);
        p.setY (4);
        System.out.println( p.getX() );
        System.out.println( p.getY() );
    }
};
```

2. Constater qu'on ne peut appeler aucune méthode sur un objet `Point` sans avoir initialisé par

   ```
   new Point();
   ```

   (constructeur déjà défini en Java et *hérité* de la classe `Object`).

3. Remarquer que dans les méthodes on peut aussi écrire `this.x` ou `this.y` au lieu de `x` ou `y` tout court.

4. Dans votre programme de test *main*, essayer d'accéder directement à *x, y* par la notation `p.x` et `p.y` où `p` est une variable `Point`. Conclusion.

5. Dans la classe, mettre *x, y* `public` au lieu de `private` cette fois-ci. Conclusion. (Remettre ensuite *x, y* privés.)

I.2 Autres méthodes

1. Rajouter à la classe `Point` d'autres méthodes de votre choix, par exemple
   - déplacer un point d'une longueur sur l'axe des x et des y,
   - ramener un point à l'origine par une méthode *reset*(), etc.)

2. Définir aussi une fonction `public double distance (Point b)` qui calcule la distance entre le point `this` et le point `b` en paramètre. Tester en calculant la distance en les points (1,2) et (2,3).

3. Définir aussi une fonction (version-2) `public static double distance (Point a, Point b)` qui calcule la distance entre les points *a* et *b* en paramètres. Comment utiliser cette méthode? Quelle est la différence avec le cas 2 précédent? Quelle serait votre choix de conception d'une fonction distance, le cas 3. ou 2. ?

I.3 Egalité ou pas entre deux objets?

1. La fonction `public boolean equals(Object o)`, héritée aussi de `Object`, permet de tester «l'égalité» entre deux objets. Usage: *p* et *q* étant deux instances de `Point`.

   ```
   p.equals(q);
   ```

   permet de tester si les deux points `p` et `q` sont égaux ou pas.

2. Créer, toujours dans votre programme de test, deux points `p` et `q` et leur affecter les mêmes coordonnées. Vérifier le résultat de la méthode `equals` sur ces points. Conclusion? (La réponse est `false`, voir ci-après)

3. Même question, mais cette fois-ci, le point *p* est initialisé normalement par *new* et *q* est initialisé par l'affectation

   ```
   q = p;
   ```

   Conclusion?

   Réponse: en 2. on compare en fait les *références* de deux points *différents* (même si ayant même valeurs).

   En 3. on compare deux *références égales*, car référence à un même objet.

4. Passer outre la méthode `equals` héritée et la redéfinir soi-même dans la classe `Point` comme suit:

   ```
   public boolean equals(Point a){
         return (this.x==a.x && this.y==a.y);
   }
   ```

   où on compare deux points par leur coordonnées. Refaire le test précédent. Conclusion? (Voir la réponse(*) en fin de TD)

II. Documenter la classe

1. Rajouter des commentaires pour documenter votre classe et vos méthodes à la façon déjà donnée par `/** ....*/` NB. Définir les commentaires des méthodes juste avant la méthode.
2. Créer un sous-répertoire (appelé `docPoint` par exemple) et faire ensuite la commande:

   ```
   javadoc -d docPoint Point.java
   ```

3. Les fichiers nécessaires à la documentation de la classe `Point` seront créés dans le répertoire donné `docPoint` sous forme de page Web. Un fichier `index.html` est le point d'entrée dans ce répertoire. Le visualiser.

III. Notion de constructeur

Reprendre la classe `Point` avec les deux nouvelles méthodes suivantes qui sont des *constructeurs*.

```
/**
 *  Initialise un point à l'origine
 */
   public Point(){x = 0; y = 0;}


 /**
  *  Initialise un point à a et b)
  */
  public Point(int a, int b){x = a; y = b;}
```

1. Dans les programmes, *main*(), utiliser

   `Point p = new Point();` pour déclarer et initialiser un objet `Point` à (0, 0) par défaut

   `Point q = new Point(5,2);` pour déclarer et initialiser un objet `Point` à (5, 2).

   A noter:

   - Avec `new Point()` il sera fait appel au constructeur sans paramètre défini juste ci-dessus, au lieu de celui hérité comme dans §I.1.
   - **Important**! Initialiser un point par constructeur, e.g. `Point p = new Point (2,5)`, n'est pas la même chose que lui affecter des valeurs par les méthodes `p.setX(2)` et `p.setY(5)`, même si dans les deux cas l'objet a la même valeur. En effet, dans le premier cas, *initialisation*, l'objet `p` n'existe pas avant son initialisation, alors que dans le deuxième cas, *affectation*, `p` est déjà créé mais ne fait que changer de

valeurs.

2. Vérifier que si on omet le constructeur `Point()` sans paramètre, c'est une erreur de compilation. En effet, à partir du moment qu'un constructeur est déclaré, l'appel `new Point()`, ne cherche pas le constructeur défaut hérité précédemment, mais celui que l'utilisateur doit définir aussi.

3. Remplacer maintenant le code `{x = 0; y = 0;}` du constructeur défaut, par `{this (0,0);}`. Vérifier le résultat. L'instruction `this(0,0)` est un appel de l'instance courante au constructeur `Point(int a, int b)` avec ici 0 et 0 comme paramètres.

   C'est d'ailleurs *la seule fois* qu'un constructeur peut être appelé *explicitement*.

4. *Exercices*:

   a. Rajouter un constructeur avec un seul paramètre (initialisation de l'abscisse) qui affecte ce paramètre à x, et 0 à y.

   b. Ecrire le code de ce constructeur de deux façons différentes.

   c. Vérifier qu'on peut réécrire le constructeur (défaut) par `{this (0);}` qui fait appel au constructeur `public Point(int a)` nouvellement ajouté.

IV Conversion vers texte.

Comme pour la méthode `equals (Point)` (cf. I.3) , on peut redéfinir la méthode `toString()` héritée aussi de `Object`, pour convertir un objet `Point` vers une chaîne de caractères imprimable. Exemple:

```
public String toString()     {
      return "(" + x + "," + y + ")" ;
}
```

Exemple d'usage:

```
Point q = new Point (2,5);
System.out.println(q.toString());    // imprime (2,5)
```

Vérifier en changeant le message d'affichage avec un autre texte dans la méthode `toString()`.

(*): La fonction `equals` héritée de `Object` a été *redéfinie* par l'objet `Point` lui même. Cette dernière méthode est sensée passer outre ou recouvrir (*override*) la méthode héritée. C'est pour cela que deux objets `Point` distincts sont maintenant ... égaux quand ils ont *la même valeur* (mêmes cordonnées). CQFD.

Remarque: En réalité, cette solution marche quand on est sûr de comparer toujours deux objets *déclarés* `Point`. Or en Java, on peut écrire:

```
Object p = new Point (2,5);
```

aussi bien que

```
Object q = new Point (2,5);
```

Là, si on compare `p` et `q`, par `p.equals(q)`, le résultat sera `false`! Justement, la surcharge va faire choisir la méthode `boolean equals(Object)` héritée de la classe `Object`, qui compare les références, au lieu de la méthode `boolean equals(Point)` déclarée dans `Point`! Il faut donc refaire le profile de cette dernière et ensuite la récrire :

```
boolean equals(Object a){
        return (this.x == ((Point)a).x  &&
                this.y == ((Point)a).y ) ;
}
```

où on convertit explicitement `a` à `Point` avant de comparer. reCQFD.

Hum...! Mais alors, rien n'empêche d'appeler `equals` avec n'importe quel objet en paramètre `a`, par exemple `p.equals("toto");` Le compilateur ne relèvera pas l'erreur à la compilation. L'erreur sera détectée à l'exécution "`java.lang.String cannot be cast to Point...`", lors de la conversion `((Point)a)` .

Evaluation

Certains exercices seront évalués à la demande et corrigés par l'encadrant du TP.

Programmation par les Objets en Java

Najib TOUNSI
Relations entre classes en Java (Séance 4)

I. Relation d'utilisation

Reprendre la classe *Point* (avec constructeurs) du TD précédent et la compiler. Créer (nouveau fichier, `Rectangle.java`) une classe *Rectangle*, qui utilise cette classe `Point`.

```
    class Rectangle {
        private Point hg,bd ;    // Les coins haut à gauche et bas à droite
                                 // Rectangle droit

    public Rectangle(){
    // rectangle par défaut. Choisir son initialisation
    }

    public Rectangle(Point h, Point b){
    // initialisation des coins à partir des paramètres données
    }

    public void afficher(){
    // Affiche les coordonnées des coins
    }

    public int surface(){
    // calcule de la surface
    }

    public void zoom(int deltax, int deltay){
    // Dilatation des coordonnées. Delta donné.
    }
        // autres méthodes...
};
```

1. Programmer les méthodes et les tester. (Nouvelle classe `TestRectangle.java` avec méthode *main*()).
2. Rajouter à la classe `Point` un constructeur copie

   ```
   Point (Point p) {...}
   ```

   et l'utiliser dans le constructeur `Rectangle (Point h, Point b)`. Quel est l'intérêt par rapport à avant?
3. Rajouter les méthodes appropriées `set` et `get` qui modifient et lisent les champs d'un rectangle
4. Créer maintenant une classe `Fenetre` (composée d'un rectangle et d'un texte du titre). Imaginer des opérations utiles, comme le déplacement, l'agrandissement d'une fenêtre, etc. Quelle opération de la classe `Rectangle` pourrait servir pour l'agrandissement d'une fenêtre?

II. Relation d'héritage

II.1 Essai Simple Héritage: Rajout de champ et de méthode

1. Créer une sous-classe `Robot` de la classe `Point` (TD précédent) et qui consiste en un point avec un champ `direction` (entier compris entre 1 et 4) et une méthode `avancer ()` qui avance le robot dans un sens selon la valeur du champ `direction` (1 monter d'un pas, 2 à droite d'un pas, 3 descendre d'un pas et 4 à gauche d'un pas.)

```
class Robot extends Point {
    int direction;       // direction actuelle

     public Robot(){
        super (0,0);
        direction = 2;  // Est
     }
     ...
     public void avancer() {
        switch (direction) {
            case 1: y++; break;
            case 2: x++; break;
            case 3: y--; break;
            case 4: x--; break;
            default:;
        }
    }
};
```

2. Faire un programme qui instancie un `Robot`, le fait avancer et imprime ensuite les cordonnées du `Robot`. Utiliser `toString()` héritée de `Point`.
3. Rajouter une méthode `setDirection (int d)` qui change la direction d'un `Robot` à la valeur *d*. Tester.
4. Rajouter un constructeur paramétré par trois entiers (les deux coordonnées et la direction) et qui initialise un robot avec ces valeurs.

II.2 Essai Simple Héritage: Polymorphisme

1. Redéfinir pour un `Robot` la méthode `toString()` héritée de `Point`. Y rajouter aussi la direction en plus des coordonnées *x* et *y*.
2. Tester la suite d'instructions suivante:

```
Point p = new Point (6,7);
System.out.println(p.toString());
p = r;
System.out.println(p.toString());
```

3. Constater le changement *dynamique* (à l'exécution) de méthode `toString()` appelée pour `p` dans les deux cas. D'abord celle de `Point` et ensuite celle de `Robot`, selon l'instance référencée par `p`.

Programmation par les Objets en Java

Najib TOUNSI
Interfaces et Classes Abstraites (Séance 5)

Les Interfaces

Les Classes Abstraites

Classe Abstraite vs Interface

Les Interfaces

Une interface est une collection nommée de déclarations de méthodes (sans les implémentations). Une interface peut aussi déclarer des constantes.

Exemple: Une interface simple avec une constante et une fonction non `static`.

```
interface Loisir {
    public int distance = 21;
    public void courirOuMarcher();
}
```

- Noter le mot **`interface`** au lieu de `class`.
- Une interface ne peut pas avoir de méthodes déclarées `static`.
- Ne peut pas avoir de méthodes implémentées non plus (à la différence des classes abstraites, voir plus loin).
- *distance* considérée comme `final variable` et ne peut être modifiée.

L'implémentation d'une interface, c'est à dire des méthodes déclarées, doit être faite dans une classe séparée. Voici un exemple de classe qui implémente l'interface ci-dessus :

```
class Coureur implements Loisir {
  //Implémentation de la méthode courirOuMarcher
  public void courirOuMarcher(){
        System.out.println("Je cours "+distance+" Km.");
    }
};
```

Noter le mot clé ***implements***.

Exemple d'utilisation:

```
class Test{
    static public void main(String args[]){
        Coureur c = new Coureur() ;
        c.courirOuMarcher();
    }
}
```

Compiler l'interface et ensuite ces deux classes. Exécuter le teste.

Résultat :

```
Je cours 21 Km.
```

Exercices :

1. Peut-on déclarer la variable de type `Coureur` et l'instancier avec avec un objet `MonImplement`? C'est à dire

   (a) `Loisir c = new Coureur();` au lieu de
   (b) `Coureur c = new Coureur() ;`

   Vérifier que c'est possible.
   Vérifier que `Loisir l = new Loisir()`, par contre, n'a pas de sens. On n'*instancie pas une interface*, car les méthodes n'y sont pas définies. Ne pas confondre avec la ligne (a), où l'instance est un objet `Coureur`.

2. Rajouter à la classe `Coureur` la méthode

   ```
   public  void courirMoins() {
        System.out.println("Je cours "+(distance/2)+" Km.");
     }
   ```

   et dans *main* l'appel

   ```
   c.courirMoins();
   ```

   Vérifier cet appel dans les deux cas (a) et (b) ci-dessus (rajouter cet appel dans `main`).

   Réponse: Erreur de compilation dans le cas où `c` est de type `Loisir`. Normal, la méthode `courirMoins` n'est pas déclarée dans l'interface.

3. La constante distance est `final`, et ne peut être modifiée ni dans la classe `Coureur` ni ailleurs. Le vérifier en faisant :

   ```
   distance /= 2;
   ```

   dans la méthode `CourirMoins`.

   Ailleurs, l'accès à la constante, en lecture donc, peut se faire soit par `c.distance` soit par `Loisir.distance`tout simplement.)

4. Créer une autre classe `Marcheur`, qui implémente la même interface `Loisir`.
   L'implémentation affichera un simple message "`Moi, je marche...`".

5. Faire un programme qui crée un tableau `mesLoisirs` d'objets `Loisir`, et qui l'instancie indifféremment avec des objets de type `Coureur` ou `Marcheur`. Vérifier que les appels à la méthode `coureurOuMarcher` sur les éléments du tableau donne le message correspondant chaque objet. (Se contenter d'un tableau deux éléments.)
   Cela rappelle le polymorphisme d'héritage. La bonne méthode exécutée en fonction de l'instance actuelle.

6. Concevoir deux interfaces *A* et *B* et une classe *C* qui implémente ces deux interfaces.

   syntaxe: `class C implements A, B {. corps de la classe..}`

7. Que se passe t-il si les deux interfaces *A* et *B* déclarent une même méthode *f*()? Une même

constante *x*?

Indication: pour le savoir, créer une classe *Test* qui utilise *f*() et *x*.

Pourquoi ne peut-on utiliser *x*? (réponse: La définition de *x* est donnée *dans* les interfaces **A** et *B*. Celle de *f*() non, elle est dans la classe C qui implémente. Il faudrait préfixer x par le nom de l'interface.).

Les Classes Abtraites

Une classe abstraite est une classe qui peut contenir des méthodes sans corps, dites méthodes abstraites. L'implantation est laissée (déléguée) aux futures sous classes de la classe abstraite.

Une classe abstraite ne contient pas forcément des méthodes abstraites. Le fait qu'une classe soit abstraite, implique qu'on ne peut pas en créer des instances. Il faut alors en dériver des sous classes pour pouvoir instancier des objets et leur appliquer des méthodes.

Mais une classe qui contient une méthode abstraite doit être déclarée abstraite. De même, une sous-classe qui ne fournit pas l'implantation d'une méthode abstraite déclarée dans une classe mère, doit être déclarée abstraite à son tour.

Exemple:

```
abstract class Generale {

     public int x=2;         // x variable d'instance (non considérée static)

     abstract public void qui();  // methode abstraite à implementer par les sous-classes

     public void moi(){
         System.out.println("Methode générale");
        }
}
```

NB. Une méthode sans corps doit toujours être déclarée abstraite. Par ailleurs, une méthode abstraite ne peut être déclarée `static` (pourquoi?)

Implantation :

```
class Speciale1 extends Generale{

    public  void qui() {
       // Implementation de qui()
        System.out.println("C'est la sous-classe Speciale1");
    }
}
```

Noter `extends` au lieu de `implements`.

Test :

```
class Test{
    static public void main(String args[]){

        Speciale1 o = new Speciale1();
        o.moi();
        o.qui();
        o.x++;
        System.out.println(o.x);
```

```
    }
}
```

NB. Compiler d'abord la classe `Generale`.

Résultat obtenue :

```
$ java Test
Methode générale
Methode concrete qui dans sous-classe Speciale1
3
```

Exercices:

1. Ajouter dans la classe `Speciale1` une redéfinition de la méthode moi déjà définie dans la classe `Generale` (afficher un message approprié). Refaire le teste pour vérifier que ce message s'affiche la place de "`Methode générale`"
2. Faire une hiérarchie de plusieurs classes abstraites. Vérifier les règles énoncées précédemment (§ classes abstraites), à savoir l'implémentation d'une méthode abstraite peut être différée d'une sous classe à une autre, par exemple.

Classe Abstraite vs Interface

- Une interface ne peut implanter une méthode alors qu'une classe abstraite peut.
- Une classe peut implanter plusieurs interfaces mais ne peut avoir qu'une seule superclasse.
- Une interface n'appartient pas à la hiérarchie des classes issues de `Object`. Des classes sans aucun rapport entre elles peuvent implanter la même interface.
- Plus important, avec `Interface` on est sûr qu'un objet a implémenté une méthode (il ne peut différer cette implantation).
- Une interface peut faire *extends* de +r autres interfaces
  (Le vérifier sur un exemple `interface C extends A , B {}`)

En réalité, à part les différences de forme, une interface est une *spécification* (abstraite) appelée à être implémentée (concrétisée) de *plusieurs façons* par plusieurs classes différentes, chacune à sa manière. Le fait, est que ces classes sont quelconques et n'ont aucune relations entre elles. C'est une vision génie logiciel plus générale.

Dans le cas des classes abstraites, les classes qui implémentent l'abstraction *sont* des sous-classes. C'est plus une vision classification et développement progressif et structuré.

> **Extrait de la documentation Sun** *Implementing an interface allows a class to become more formal about the behavior it promises to provide. Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler. If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.*
>
> (http://docs.oracle.com/javase/tutorial/java/concepts/interface.html)

That's all folks

Travaux pratiques en Java

Séance 6: Copies d'objets

Najib Tounsi

1. Le Clonage

2. Simple clonage

3. Clonage en profondeur

1. Le Clonage

Le clonage en général est la création d'un nouvel objet (une instance d'une classe) à partir d'une instance déjà existante. En java le clonage sert à faire la copie d'un objet dans un autre. En effet, si on se contente d'écrire $x = y$; on obtient la copie des références (y sur x), et non la duplication de l'objet y dans x.

Pour qu'un objet soit copiable (*clonable*), sa classe doit implanter la méthode `clone()` de l'interface `Clonable`. En fait, la méthode `clone()` est définie pour la classe `Object` et s'applique donc à tous les objets Java. En pratique, les sous-classes qui le désirent doivent implanter l'interface `Clonable` et redéfinir la méthode `clone()`. Cette redéfinition peut se limiter à faire appel à `super.clone();` i.e. la méthode mère héritée. Celle-ci peut générer l'exception `CloneNotSupportedException`.

2. Simple clonage

Exemple 2.1. Création d'un objet Cellule suivi de son clonage

Soit la classe Cellule, constituée d'un entier et d'un tableau:

```
class Cellule extends Object implements Cloneable {
 // Doit implementer la méthode clone() de l'interface Clonable.
 //   Methode qui fait le clonage.

    // Donnees
    int i = 0;
    int[] t = {1, 2};

    // Méthodes
    public Object clone(){
        try {
            return super.clone();
        }
        catch (CloneNotSupportedException e){
            throw new InternalError();
        }
    }

    public void afficher(){
        System.out.println(i +" "+ t[0]+" "+t[1]);
    }
}
```

Tester cette classe avec:

```
class TestClone {
     public static void main(String args[]){
        Cellule x = new Cellule();        // x Objet Cellule
        x.afficher();
        Cellule y = (Cellule) x.clone();  // y clone de x
        y.afficher();
     }
}
```

et constater le résultat:

```
i=0 t=(1, 2)  // valeurs de la cellule x
i=0 t=(1, 2)  // valeurs de la cellule y
```

Remarquer qu'on aurait pu faire une affectation d'objets dans `main()`. Au lieu de faire

```
   Cellule y = (Cellule) x.clone();   // initialisation par clonage
```

On fait

```
    Cellule y = new Cellule();
    y = (Cellule) x.clone();     // vraie affectation d'un clone
```

Clonage superficiel.

En réalité, le clonage est *superficiel*. L'entier *x.i* a été copié dans *y.i*, mais pour le tableau *t*, c'est la référence qui est copiée (non les éléments du tableau.) Pour le constater, rajouter à la classe `Cellule` une méthode qui change les éléments d'un cellule.

Exemple 2.2. Rajouter à la classe `Cellule` la méthode

```
public void changeMe(){
    i = 10;
    t[0] = 11;
    t[1] = 12;
}
```

et vérifier qu'en rajoutant dans `main()` les instructions:

```
x.changeMe();
x.afficher();
y.afficher();
```

on obtient:

```
i=10 t=(11, 12)        // x change
i=0 t=(11, 12) // y aussi pour la partie tableau (même objet reference)
```

3. Clonage en profondeur

Pour copier tout l'objet `Cellule`, y compris le tableau, il faut cloner ce dernier (le copier aussi).

Exemple 3.1. Cloner aussi le tableau.

On change la méthode `clone()` de la classe `Cellule`, pour cloner le tableau (le copier aussi).

```
public Object clone(){
      try {
          Cellule tmp = (Cellule) super.clone();
          tmp.t = (int []) this.t.clone();  //clonage de this.t
          return tmp;
       }
       catch (CloneNotSupportedException e)
          {throw new InternalError(); }
}
```

On crée par clonage une cellule temporaire `tmp` et on modifie son tableau `tmp.t` par recopie (`this->t.clone()`)

Exercice: le vérifier.

Exemple 3.2. Variation sur le même thème, clonage à la C++

Cette fois-ci, on va copier un à un les éléments d'un objet `Cellule` sans faire appel à `super.clone()`. On modifie la méthode `clone()` de la classe `Cellule` comme suit.

```
public Object clone(){ // à la c++, sans clone super.clone()
        Cellule tmp = new Cellule();
        tmp.i=i;
        tmp.t = (int []) this.t.clone();     //clonage de this.t
        return tmp;
}
```

Remarque: on peut tout aussi aussi faire `for` pour copier les éléments de `t` (exercice: le faire)

Exemple 3.3. Constructeur Cellule par copie. Façon C++.

On peut créer un constructeur par copie pour la classe `Cellule`, comme en C++.

```
public Cellule (Cellule x){
     this.i = x.i;
     for (int i=0; i<2; i++)
         this.t[i]=x.t[i];
 }
```

et qu'on peut utiliser par

```
    Cellule x = new Cellule();
    ...
    Cellule y = new Cellule(x);  // initialisation par copie
```

Exercices: Le vérifier. Quelle est la différence par rapport à la méthode clone? (*réponse*: un constructeur sert à initialiser un objet nouveau, alors qu'un clonage en Java peut servir aussi à faire des affectations d'objets).

Programmation par les Objets en Java

Travaux Dirigés 7

Najib Tounsi

(Lien permanent: http://www.emi.ac.ma/ntounsi/COURS/Java/TD/tdJava7.html (.pdf))

Les interfaces comme TADs, avec plusieurs implémentations.

Interface Pile

Soit un TAD *pile de caractères*, défini par l'interface:

```
/**
 Interface Pile
*/

interface Pile {
    final int MAX = 8;

    public void empiler(char c);
        // Empile un char
    public char sommet();
        // Retourne le sommet de la pile
    public void depiler();
        // Retire le sommet de la pile
    public boolean vide();
        // Teste si la pile est vide
    public boolean pleine();
        // Teste si la pile est pleine
};
```

On peut réaliser cette pile par un chaîne de caractères ou un tableau de caractères etc.

1ère implémentation de la pile

On peut utiliser `StringBuilder` pour mémoriser les caractères d'une pile. Déclarer par exemple :

```
StringBuilder s = new StringBuilder(MAX);
```

On peut rajouter un caractère *c* à une chaîne *s* à l'endroit *i* par :

```
s.insert (i, c);
```

On peut consulter le *i*-ème caractère d'une chaîne *s* par :

```
c = s.charAt (i);
```

Exercices :

1. Utiliser ce qui précède pour créer une classe `MaPile` qui implémente une pile avec `StringBuilder`.
   Indication : empiler en rajoutant les caractères les uns derrière les autres, par exemple "b", "br", "brd" etc.
2. Ecrire un programme test qui lit des caractères et les imprime en ordre inverse. Dernier caractère lu est '`#`'.

2ème implémentation de la pile

On peut utiliser un tableau Java pour mémoriser les caractères d'une pile (c.f. TD2). Déclarer par exemple:

```
char[] t = new char[MAX];
```

L'accès au éléments du tableau se fait normalement par indice, e.g. `t[i]`.

Exercice :

1. Créer une classe `TaPile` qui implémente une pile avec un tableau cette fois-ci. (c.f. TD2).
2. Reprendre le programme test précédent pour tester cette 2e implémentation.

That's all folks