INF 220 Algorithmique

Jean-François Remm
Jean-François Berdjugin
Vanda Luengo

INF 220 Algorithmique

par Jean-François Remm, Jean-François Berdjugin, et Vanda Luengo

Date de publication 2011

Table des matières

Liste des illustrations

Liste des tableaux

Liste des exemples

Chapitre 1. Introduction

1. Le programme pédagogique national

1.1. Objectifs – Compétences Minimales

Aborder des techniques algorithmiques avancées. Appréhender les notions de complexité et de structures de données. Mettre en oeuvre des algorithmes.

Apprentissage d'un langage de programmation.

1.2. Contenu

Notion de complexité, Récursivité, Structures de données (tableaux, listes, files, piles, arbres, fichiers), Les algorithmes associés à ces structures de données (parcours, mise à jour, tri).

2. Choix et objectifs

Nous allons privilégier la pratique à la théorie ; la notion de complexité sera sommairement introduite, puis nous étudierons les tableaux et quelques tris. Avant d'aborder les autres structures de données (liste, piles, arbres) nous introduirons les notions de classes et d'objets. Un accent particulier sera porté sur les classes et les objets car ils constituent les briques de base de la programmation orienté objet qui suivra cette matière. Le langage de programmation choisi est Java™. Nous devrez à la fin maîtriser l'utilisation des tableaux, la création de classes et l'utilisation d'objets.

3. Le déroulement des séances et les évaluations

Nous disposons de trois cours magistraux dans lesquels nous aborderons les structures de données, les tableaux, la notion de complexité, les algorithmes de tri. Pour les TD, nous disposons de trois séances de trois heures que nous utiliserons pour étudier les tableaux, les séances seront mixtes à la fois sur papier et à la fois sur machines, vous travaillerez à deux par postes. Nous finirons avec neuf séances de TP de trois heures ou nous étudierons les classes et objets, les listes, les piles et les arbres.

Vous aurez deux évaluations obligatoires, une première portant sur les tableaux réalisée pendant la première séance de TP et une évaluation finale de trois heures également sur machine. Une évaluation facultative est possible sous-forme de devoir maison.

Chapitre 2. Cours

1. Structures de données

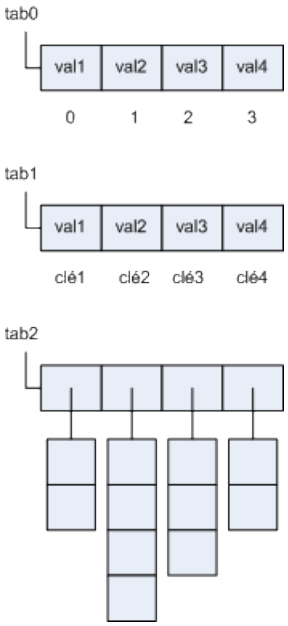
Il est possible de classer les structures de données : les structures finies (constantes, variables, enregistrements, structures composées finies), les structures indexées, les structures récursives.

Voici quelques exemples de structures de données, les cinq premières sont qualifiées de linéaires, la quatrième est qualifiée d'arborescente et enfin la dernière est qualifiée de relationnelle.

1.1. Les tableaux

Les tableaux sont la première structure de données que nous étudierons, ils correspondent à la représentation intuitive que l'on peut se faire d'un tableau : un ensemble de cases accessibles par un ou plusieurs noms.

Figure 2.1. tableaux



Les tableaux contiennent plusieurs valeurs contenues dans des cases accessibles via une clef, si les clefs sont des entiers croissant on parle de tableaux indicés, sinon de tableaux associatifs. Dans les tableaux associatifs la clef peut-être de n'importe quel type. Enfin, une case peut contenir un autre tableau, on parle alors de tableaux à plusieurs dimensions.

Les opération de base pour des tableaux indicés de taille fixe sont :

- La modification, par exemple tab0[1] =2; la case d'indice 1 reçoit la valeur 2,
- La consultation, par exemple int i = tab0[1]; la variable i reçoit la valeur de la case d'indice 1.

1.2. Les listes simplement chaînées

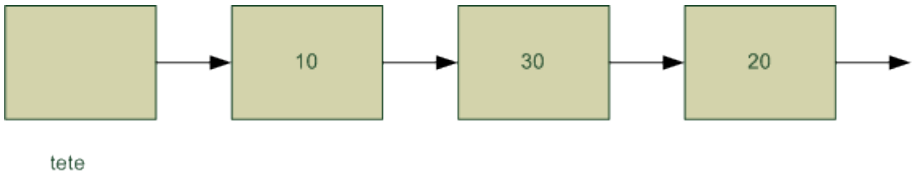
Les listes sont des structures séquentielles, elles peuvent-être implémentées sous forme de tableaux ou de sous forme chaînée.

Les opérations de base sont :

- l'insertion,
- la suppression,
- la recherche,
- la concaténation.

Les listes simplement chaînées sont constituées de maillons ou de cellules qui contiennent une valeur et une référence vers la cellule suivante.

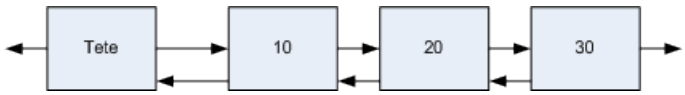
Figure 2.2. Liste simplement chaînée



1.3. Les listes doublement chaînées

La liste doublement chaînée est constituée de cellules ayant une référence vers la cellule suivante et la cellule précédente.

Figure 2.3. Liste doublement chaînée

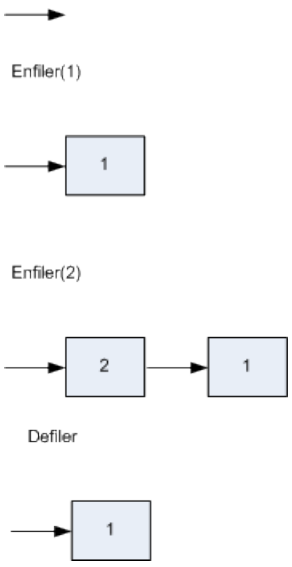


1.4. Les files

Les files possèdent deux opérations de base :

- enfiler,
- défiler.

Figure 2.4. File



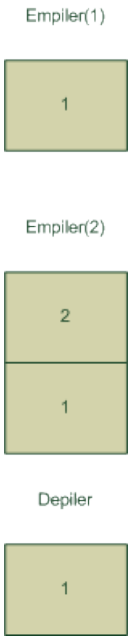
La file est qualifiée de FIFO (First In First Out). Il existe des variantes comme les files à priorité.

1.5. Les piles

La pile possède deux opérations de base :

- empiler,
- dépiler.

Figure 2.5. Pile



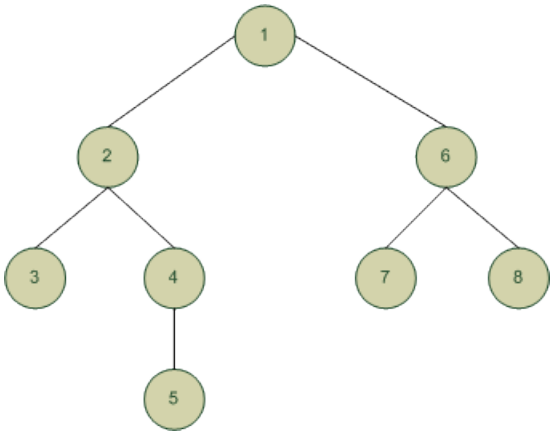
La pile est qualifiée de LIFO (Last In First Out).

1.6. Les arbres binaires

Les arbres sont constitués de noeuds, un noeud particulier est la racine. Les noeuds peuvent posséder des sous arbres.

Les arbres imposent une approche récursive.

Figure 2.6. Arbre binaire

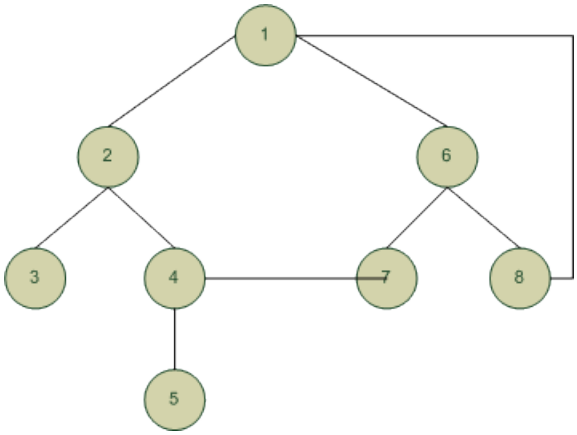


Les arbres binaires sont des sous-cas des arbres où les noeuds possèdent au plus deux fils (les fils sont les racines des sous-arbres).

1.7. Les graphes

Les graphes sont des structures où des cycles sont possibles.

Figure 2.7. Graphe



2. Complexité

Commençons par un exemple :

```
int s = 0;
int i;
for (i=0; i < n; i++)
    s=s+i;
```

Cet algorithme coûte en mémoire deux entiers, c'est son coût spatial, mais quel est son coût temporel ? Si l'on ne prend pas en compte les déclarations, que chaque opération d'écriture, de lecture, de comparaison ou de calcul coûte 1, nous avons un coup de

$$2 + 8n + 1. \tag{2.1}$$

Nous observons ainsi que le coût temporel de notre problème dépend de la taille de l'entrée (n) et des hypothèses de consommation de temps.

Voici un deuxième exemple :

```
int i = 0;
for (i=0; i <n; i++)
    if (a[i]>i)
       a[i] = a[i] -1;
```

Cette fois-ci le coût temporel de l'algorithme est plus difficile à calculer, c'est pourquoi nous allons nous intéresser au plus mauvais temps d'exécution. Il aussi possible d'avoir le meilleur temps d'exécution et le temps d'exécution moyen.

Nous avons

$$T_{min}(n) \leq T_{moyen}(n) \leq T_{max}(n). \tag{2.2}$$

Maintenant comment choisir le bon algorithme, est-il besoin du calcul du plus mauvais temps de calcul ($T_{max}(n)$) ou un ordre de grandeur peut-il suffire.

Par exemple, si nous avons *2 + 8n +1* et *3 + 4n²+ 5*, le premier algorithme a un ordre de grandeur en *n* et le second en n^2.

2.1. Définition de l'ordre de grandeur asymptotique

L'ordre de grandeur asymptotique, la notation O (grand O) permet la comparaison entre les temps d'exécution. O est défini comme suit :

Soit f(n) et T(n) deux fonction de N vers R*+, T=O(f) (T est en grand O de f) ssi il existe c appartenant à R*+, il existe n_0 tq quelque soit $n > n_0$, $T(n) \leq c\ f(n)$ (2.3)

Donc 2 + 7n +1 est en O(n) et 3 + 4 n^2 +5 est en O(n^2).*Pour être plus formel, il nous faudrait une autre notion le grand Θ qui permet de définir des équivalences, par exemple 2n est en Θ(n) mais n'est pas en Θ(n^2).*

2.2. Exemples de complexités courantes

Le tableau suivant contient les complexités courantes et les valeurs pour n =1000.

Tableau 2.1. Complexités courantes

O	Dénomination	Valeur pour n=1 000 000
O(1)	constant	
O(ln n)	logarithmique	13.81
O(n)	linéaire	1 000 000
O(n ln n)	n log	13 815 510
O(n^2)	quadratique	1 0000 000 000 000
O(n^3)	cube	1 0000 000 000 000 000 000
O(2^n)	exponentiel	...

2.3. Règles de simplification

Il est possible de simplifier le temps d'exécution pour les ordres de grandeur :

1. Les constantes n'ont pas d'importance, T(n) = n + c => O(n)
2. Les coefficients n'ont pas d'importance, T(n) =c n => O(n)
3. Dans un polynome, les termes d'ordre inférieurs n'ont pas d'importance, T(n) = $2^n + n^{1000}$ = O(2^n).

3. Algorithme de tri

Les tris peuvent-être internes ou externe (il utilisent d'autre structures de données), séquentiels ou parallèles. Nous allons aborder des algorithmes de tri internes, séquentiels avec une approche itérative.

La compléxité des tris pour des données quelconques ne peut-être inférieur à *n ln* pour les tris par comparaison et est souvant en n^2. Les algorithmes qui suivent sont dits lents, leur complexité est en n^2.

3.1. Tri par insertion

L'idée est de prendre les éléments les uns après les autres et de les insèrer parmi les éléments déjà triés => Une boucle pour parcourir le tableau, une boucle pour insérer à la bonne place. La complexité est en n^2.

```
public static void triInsertion(int tableau[])
    {
  int i,j, m;
```

```
  int l = tableau.length -1;
  for (i=1; i <= l; i++)
  {
   m = tableau[i];
   j=i;
   while (j>0 && tableau[j-1]>m)
   {
    tableau[j]=tableau[j-1];
    j=j-1;
   }
   tableau[j]=m;
  }
    }
```

3.2. Tri par selection

L'Idée est de trouver le plus petit et le placer en premier puis trouver le plus petit parmi les éléments non placés et le placer en second, etc. => Une boucle de parcour du tableau Une boucle pour trouver le minimum. La complexité est en n^2.

```
public static void triSelection(int tableau[])
 {
  int i,j,m, min;
  int l = tableau.length - 1;
  for (i=0; i< l; i++)
  {
   min = i;
   for (j=i+1; j<=l; j++ )
   {
    if (tableau[j]< tableau[min])
     min=j;
   }
   m=tableau[i];
   tableau[i] = tableau[min];
   tableau[min] = m;
  }
 }
```

3.3. Tri à bulle

L'iIdée est de permuter les éléments adjacents mal placé et de parcourir autant de fois que nécessaire le tableau => Une boucle pour selectionner les éléments et une boucle pour les permutations. La complexité est en n^2.

```
public static void triBulle(int tableau[])
 {
  int i,j,m;
  int l = tableau.length -1;
  for ( i=l; i>=0; i--)
   for (j=1; j<=i; j++)
    if (tableau[j-1] > tableau[j])
    {
     m = tableau[j-1];
     tableau[j-1] = tableau[j];
     tableau[j]=m;
    }
 }
```

4. Programmation orientée objet

La programmation orientée objet (POO) est un paradigme de programmation où les briques logicielles sont des objets. Les objets représentent des concepts, des entités du monde physique, comme un point, un véhicule, ... La communication entre objets via leur relations permet de réaliser les fonctionnalités du système (de l'application). Il existe deux catégories de langages objets : les langages à classes comme Java et les les langages à prototypes comme JavaScript. Nous étudirons, cette année, le premier type, les classes définissent la structure interne des objets et les messages auxquels ils répondent.

Figure 2.8. Diagramme de classe de la classe Point

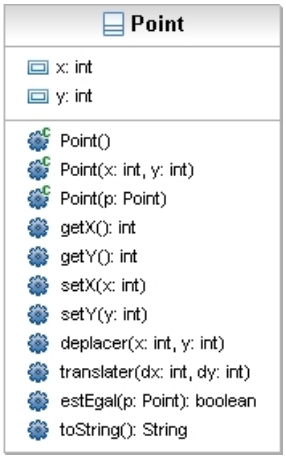
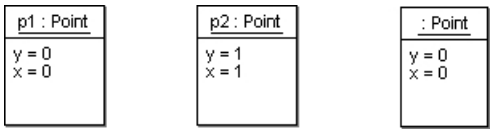


Figure 2.9. Objets ou instances de la classe Point



4.1. l'objet

L'objet est une structure de données, c'est pourquoi nous l'étudions maintenant. Un objet rassemble dans une même entité deux éléments de la programmation procédurale :

les attributs ou champs ou variable d'instance
: ils définissent l'état de l'objet

les méthodes
: elles définissent le comportement de l'objet en utilisant et ou modifiant les variables d'instances. Elle correspondent aux messages échangés entre objets.

4.2. La classe

Les objets sont dynamiques et les classes statiques, les objets existent en mémoire et sont des instances des classes, ainsi les classes sont des moules et les objets les réalisations. La classe décrit la structure interne et les méthodes qui s'appliqueront aux objets de même famille (même classe). La classe offre des méthodes qui permettent de créer et de détruire les objets[1].

4.3. Les fondamentaux de la POO

Nous reviendrons longuement sur ces fondamentaux en programmation, ils sont ici à titre indicatif :

l'encapsulation
: La réunion dans la même entité des données et de leurs moyens de transformation mais aussi la possibilité de rendre visible ou non tout ou partie des variables d'instances et des méthodes. Nous utliserons deux visibilités private, visible uniquement dans la classe et public visible partout.

l'héritage
: L'héritage est un moyen de définir de nouvelles classes à partir de classes existantes pour en réaliser une spécialisation.

[1] En Java, il n'y a pas d'appel explicite au destructeurs, un objet est détruit par le Garbadge Collector, si plus aucune référence ne lui est associée.

le polymorphisme
: La capacité pour un objet de changer de type à l'exécution et ainsi de choisir dynamiquement la méthode qui correspond au type réel de l'objet.

Chapitre 3. Travaux dirigés

Lors des séances qui vont suivre nous commencerons par une présentation orale, puis une mise en oeuvre sur machine.

1. Syntaxe java

Les tableaux JAVA ™ correspondent à la représentation intuitive que vous vous faite d'un tableau ou d'un vecteur. Les éléments cases sont accessibles via un indice, dont la numérotation commence à 0. La première case d'un tableau est numérotée 0 est la dernière la longueur du tableau moins 1. Les tableaux JAVA ™ sont indicés et de taille fixe*Le langage JAVA est fourni avec d'autre types de données nommées collections qui offrent plus de souplesse que les tableaux*.

Pour utiliser les tableaux, il vous maîtriser les concepts suivants :

- La déclaration
- L'instanciation ou création
- L'accès aux donnée

1.1. La déclaration

Pour déclarer un tableau les crochets*[] doivent être utilisés.*

typeTab[] tab; déclare *tab* comme étant une variable référencent un tableau de type *typeTab*. Tous les éléments de notre tableau sont de même type.

```
typeTab[] tab; //declaration de tab comme étant un tableau
               //de type typeTab
```

Figure 3.1. Déclaration d'un tableau

Ainsi `int[] tab` déclarer `tab` comme étant une variable référencent un tableau contenant des entiers, à ce stade le tableau n'existe pas encore, il faut le construire.

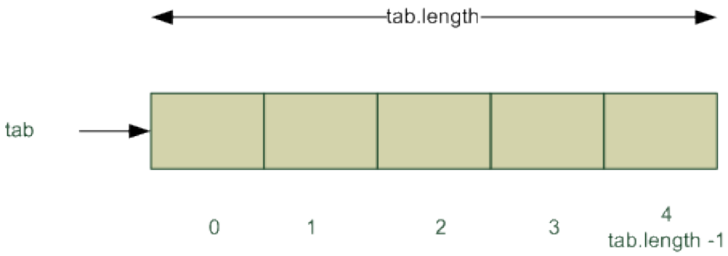
1.2. Instanciation (ou construction)

Après la déclaration notre tableau n'existe pas, nous ne disposons que de la référence (le nom), pour qu'il existe, il faut l'instancier (construire). Les tableaux ont une taille fixe (structure de donnée statique), celle taille doit être fixée lors de la construction. Le mot clef permettant la construction est `new`.

tab = new typeTab[taille]; permet de créé le tableau de type *typeTab*, de taille *taille* est accessible via la référence *tab*.

```
typeTab[] tab;

tab = new typeTab[taille]; //construction d'un tableau de taille : taille
                           // la première case est numérotée 0
                           // la dernière case est numérotée taille - 1
```

Figure 3.2. Construction d'un tableau




```
double[] tab;
//declaration de tab comme étant un tableau de réels

tab = new double[5];
//construction d'un tableau de 5 cases contenant des rééls
// et affectation du tableau à tab
//la première est indicée par 0
//la dernière est indicée par 4
```

Le tableau créer nous pouvons accèder à son contenu en lecture et écriture.

1.3. L'accès aux données

Sous un même nom, la référence, nous avons un ensemble de données accessible en utilisant un indice.

1.3.1. Lecture/Ecriture

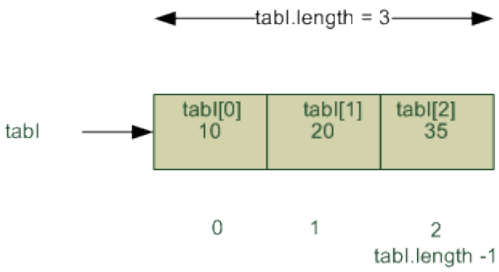
La lecture et l'écriture sont réalisées en spécifiant entre crochet (*[]*) la position de la "case".

tab[i] permet d'accéder à la position *i* du tableau référence par *tab*.

Voici un exemple d'écriture suivi d'un exemple de lecture sur un tableau d'entiers:

```
int[] tabI;
tabI = new int[3];

tabI[0]=10; //ecriture de la valeur 10 à la position 0
tabI[1]=20; //ecriture de la valeur 20 à la position 1
tabI[2]=35; //ecriture de la valeur 35 à la position 2

int i = tabI[0] + tabI[1] + tabI[2]; //i recoit la somme des elements du tableau
```

Figure 3.3. Accès aux éléments d'un tableau



C'est parfois pratique mais un tableau peut-être déclaré, construit et initialisé en une seule étape :

```
int[] t={0,1,4,9};
```

Le code précédent est équivalent à :

```
int[] t;
t= new int[4];
t[0]=0;
t[1]=1;
t[2]=4;
t[3]=9;
```

1.3.2. Propriétés

Nous allons utiliser comme propriété du tableau sa longueur : *length*.

Nous le verrons plus tard mais l'accès au propriétés d'un objet se fait en utilisant l'opérateur . (point). Ainsi la longueur du tableau *tab* est accessible en utilisant *tab.length*. Le code suivant permet d'afficher la taille du tableau *tabI* ainsi que le dernier et le premier élément.

```
int[] tabI;
tabI = new int[3];
tabI[0]=10;
tabI[1]=20;
tabI[2]=35;

System.out.println(tabI.length);
//affiche la longueur du tableau ici 3
System.out.println(tabI[tabI.length - 1])
//affiche la valeur de la derniere case du tableau ici 35
System.out.println(tabI[0])
//affiche la valeur de la première case du tableau ici 10
```

1.3.3. Pièges

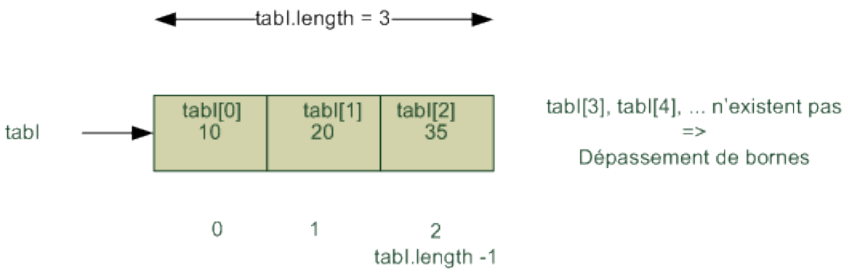
Les tableaux contiennent deux pièges intrinsèques : le dépassement des bornes et les alias involontaires.

1.3.3.1. Dépassement des bornes

Les tableaux sont des structures statiques dont la taille est choisie à la création, il est impossible d'accéder à un élément à l'extérieur des bornes. Si notre tableau *tabI* a pour taille 3 : t[-1] et t[3] conduisent par exemple au message *ArrayIndexOutOfBoundsException* (Erreur de dépassement des bornes par l'indice du tableau).

Vous rencontrez probablement se problème un jour ou l'autre aussi n'en oubliez pas la raison.

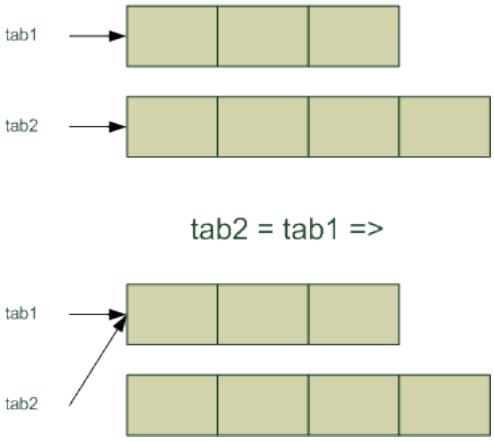
Figure 3.4. Dépassement des bornes



1.3.3.2. Alias

Nous accédons aux tableaux via une référence aussi si *tab1* et *tab2* sont deux références de tableaux : *tab1* = *tab2* ne recopie pas le tableau référencé par *tab2* dans celui référencé par *tab1* mais recopie la référence de *tab2* dans *tab1*. *tab1* devient un alias sur *tab2*, ils référencent le même tableau et toute modification par l'un affecte en conséquence l'autre. Nous avons deux noms (*tab1*, *tab2*) pour une même chose (un même tableau).

Figure 3.5. Alias



2. Mise en place

Vous aller créer sur votre répertoire partagé (`z:`) un répertoire nommé `workspace_inf220`. Puis lancer eclipse en choisissant ce répertoire comme workspace.

Dans ce workspace vous aller créer un projet nommé `inf220`.

Dans le projet `inf220` vous aller céer un paquetage nommé `td` et dans ce paquetage deux paquetage `td.mesClasses` et `td.mesTests`.

3. Exercices

3.1. Exercice 1

Donner sur papier le resultat de l'exécution du code suivant , puis coder le dans `td.mesTests` :

```
public class Test1 {

 public static void main(String[] args){
     int[] carre;
     int i;
     carre = new int[4];
     carre[0] = 2;
     carre[1] = 5;
     carre[2] = 3;
     carre[3] = 10;
     for(i=1; i< carre.length; i++){
       carre[i] = carre[i] * carre[i];
     }
     for(i=0; i < carre.length; i++){
       System.out.println(carre[i]);
     }
   }

}
```

3.2. Exercice 2

Donner sur papier le resultat de l'exécution du code suivant , puis coder le dans `td.mesTests` :

```
public class Test2 {
```

```
 public static void main(String[] args){
     int[] nb;
     int i;
     nb = new int[6];
     nb[0] = 1;
     for(i=1; i< nb.length; i++){
       nb[i] = nb[i-1] + 2;
     }
     for(i=0; i < nb.length; i++){
       System.out.println(nb[i]);
     }
   }

}
```

3.3. Exercice 3

Donner sur papier le resultat de l'exécution du code suivant , puis coder le dans `td.mesTests` :

```
public class Test3 {

 public static void main(String[] args){
     int[] suite;
     int i;
     suite = new int[6];
     suite[0] = 1;
     suite[1] = 1;
     for(i=2; i< suite.length; i++){
       suite[i] = suite[i-1] + suite[i-2];
     }
     for(i=0; i < suite.length; i++){
       System.out.println(suite[i]);
     }
   }

}
```

3.4. Exercice 4

Donner sur papier le resultat de l'exécution du code suivant , puis coder le dans `td.mesTests` :

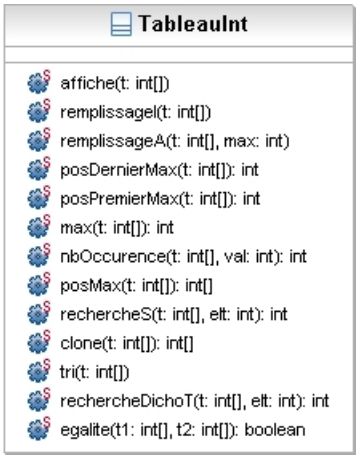
```
public class Test4 {

 public static void main(String[] args) {
  double[] t1;
  double[] t2;
  t1 = new double[3];
  t2=t1;
  t1[0]=1.1;
  t2[1]=2.2;
  t2[2]=3.3;
   for(int i=0; i < t1.length; i++){
        System.out.println(t1[i]);
      }
 }

}
```

3.5. Exercice 5

Dans l'exercice précédent `t1` et `t2` étaient deux noms vers un même tableau, on les qualifie d'alias. Ce sont les alias qui permettent en java de modifier un tableau passé en paramètre dans une méthode.

Créer dans td.mesClasses une classe `TableauInt` et dans td.mesTests une classe `TestTableauInt`.

Figure 3.6. Diagramme de classe de la classe TableauInt



Nous allons dans la suite implémenter l'ensemble de méthodes présentés dans le diagramme de classe. Toutes les méthodes serons également testés, avec la méthode `main`, dans la classe `TestTableauInt`. Comme vous constaterez dans la suite, il faut bien manipuler les boucles (for, while, do). Il faut donc bien les comprendre avant de les utiliser. Faites attention à bien choisir les bornes quand vous parcourez les tableaux (voir point Section 1.3.3.1, « Dépassement des bornes »).

3.5.1. L'affichage du tableau

Ecrire dans `TableauInt` une méthode public static void affiche(int [] t) qui affiche un tableau reçu en paramètre.

Tester votre méthode avec le `main` de `TestTableauInt`.

3.5.2. Le remplissage intéractif d'un tableau

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

permet de créer un objet *sc* de type *Scanner* qui écoute sur le clavier. Utiliser cet objet dans la méthode de classe suivante pour initialiser un tableau d'entier :

```
public static void remplissageI(int [] t)
```

Tester votre méthode.

3.5.3. Le remplissage aléatoire d'un tableau

La classe *Math* fournie une méthode *random(...)* qui permet d'obtenir un nombre réel aléatoire. Le nombre aléatoire obtenu est un double, vous pouvez pour le convertir en entier utiliser le transtypage : `(int) (Math.random() * max)`.

```
int i = (int) 1.22; //ici la valeur de i sera 1
```

Écrire une méthode de classe *remplissageA* qui prend en paramètres un tableau et une valeur maximum :

```
public static void remplissageA(int [] t,int max)
```

3.5.4. Somme des éléments d'un tableau

Écrire une méthode de classe *somme* qui prend un tableau d'entier en paramètre et permettant d'obtenir la somme des élément de ce tableau. Faites appel à cette fonction depuis avec un tableau dimensionné et initialisé. La signature de la méthode est :

```
public static int somme(int[] t)
```

Cette méthode renvoie la somme calculée.

3.5.5. Position du dernier des maximums

Écrire une méthode de classe *posDernierMax* qui prend un tableau d'entiers en paramètre et détermine la dernière position du plus grand élément de celui-ci. La signature de la méthode est :

```
public static int posDernierMax(int[] t)
```

3.5.6. Position du premier des maximums

Écrire une méthode de classe *posPremierMax* qui prend un tableau d'entiers en paramètre et détermine la première position du plus grand élément de celui-ci. La signature de la méthode est :

```
public static int posPremierMax(int[] t)
```

3.5.7. Valeur du maximum

Écrire une méthode de classe *max* qui prend un tableau d'entiers en paramètre et détermine la valeur maximun contenue dans le tableau. La signature de la méthode est :

```
public static int max(int[] t)
```

Vous pouvez utiliser une des méthodes précédentes.

3.5.8. Nombre d'occurences

Écrire une méthode de classe *nbOccurence* qui prend en paramètres un tableau d'entiers et une valeur puis qui détermine le nombre d'occurrences (le nombre d'apparitions) de la valeur dans le tableau. La signature de la méthode est :

```
public static int nbOccurence(int[] t, int val)
```

3.5.9. Position des maximums

Écrire une méthode de classe *posMax* qui prend un tableau d'entiers en paramètre et détermine les position des plus grands élément de celui-ci. La signature de la méthode est :

```
public static int[] posMax(int[] t)
```

Pour réaliser cet exercice vous devez :

1. trouver le maximum,
2. compter le nombre d'occurence du maximum,
3. créer le tableau qui sera renvoyé,
4. remplir le tableau nouvellement créé,
5. renvoyer ce dernier tableau.

3.5.10. Recherche séquentielle dans un tableau non trié

Écrire une méthode de classe *rechercheS* qui prend en paramètre un tableau d'entier et une valeur entière et qui renvoie la position (première trouvée) de la valeur ou -1 si cette valeur n'a pas été trouvée dans le tableau. La signature de la méthode est :

```
public static int rechercheS(int[] t, int elt)
```

3.5.11. Recopie d'un tableau

Écrire une méthode de classe *clone* qui permet la recopie d'un tableau. La signature de la méthode est

```
public static int[] clone(int[] t)
```

3.5.12. Tri

Écrire une méthode qui permet de trier un tableau, vous pourrez utiliser l'algorithme de tri de votre choix. La signature de la méthode est :

```
public static void tri(int [] t)
```

Vous pouvez utiliser une des méthodes vues en cours.

3.5.13. Recherche dichotomique dans un tableau trié

Écrire une méthode de classe *rechercheDicoT* qui effectue une recherche dichotomique dans un tableau trié. Cette méthode renvoie -1, si l'élément ne peut être trouvé, sinon elle renvoie sa position.La signature de la méthode est :

```
public static int rechercheDichoT(int [] t, int elt)
```

La recherche dichotomique nécessite un tableau trié, l'idée et d'utiliser deux marqueurs l'un sur la première case du tableau, l'autre sur la dernière puisde regarder la case du milieu :

- si la valeur est bonne la position est connue
- si la valeur de la case du milieu est supérieure à la valeur recherchée, il faut déplacer le marqueur supérieur au milien moins un
- si la valeur de la case du milieu est inférieure à la valeur recherchée, il faut déplacer le marqueur supérieur au milien plus un

Si les deux marqueurs se croisent, c'est que la valeur n'est pas présente dans le tableau.

3.5.14. Égalité entre deux tableau

Écrire une méthode de classe `egalite` qui permet de savoir si deux tableaux sont égaux[1]. La signature de la méthode est :

```
public static boolean egalite(int [] t1, int[] t2)
```

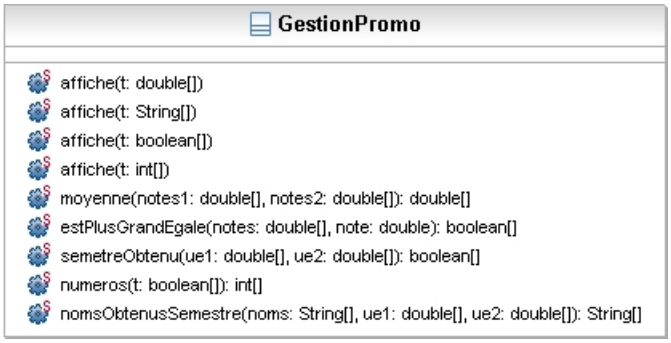
3.6. Exercice 6

Nous allons aider un enseignant à gérer sa classe, il va stocker le nom des étudiants dans un tableau de chaînes de caractères, les notes de l'UE1 dans un tableau de réels et les notes de l'UE2 dans un autre tableau de réels. Les trois tableaux ont bien entendu la même taille. Par exemple nous pouvons avoir :

```
String[] nom ;
double[] ue1, ue2 ;
nom = new String[4] ;
ue1=new double[4] ;
ue2=new double[4] ;
nom[0] = "NADRE"; ue1[0]=8 ; ue2[0]=8 ;
nom[1] = "ANDY"; ue1[1]=12 ; ue2[1]=7.5 ;
nom[2] = "KATAE"; ue1[2]=18 ; ue2[2]=8 ;
nom[3] = "ATLAIT"; ue1[3]=12 ; ue2[3]=14 ;
```

Nous allons créer une classe GestionPromo et une classe TestGestionPromo avec un main.

Figure 3.7. Diagramme de classe de la classe GestionPromo



[1] Deux tableaux de tailles différentes ne sont pas égaux.

3.6.1. affichages

Les quatres premières méthodes, similaire à celle du tableau d'entiers, permet d'afficher un tableau de rééls, un tableau de chaîne de caractère, un tableau de booléens et un tableau d'entiers. Leurs signatures sont `public static void affiche(double[] t)`, `public static void affiche(String[] t)`, `public static void affiche(boolean[] t)` et `public static void affiche(int[] t)`.[2].

3.6.2. Moyenne

Cette méthode permet à partir des notes de l'UE1 et des notes de l'UE2 de réaliser la moyenne des deux UE. Notre méthode prendra deux tableaux de notes (notes1 et notes2) et retournera un tableau de notes : `public static double[] moyenne(double[] notes1, double[] notes2)`. On suppose que notes1 et notes2 ont la même taille. Sur notre exemple nous aurions pour moyenne(ue1, ue2) comme résultat le tableau : {8, 9.75, 13, 13}.

3.6.3. estPlusGrandEgale

Nous souhaitions écrire une méthode qui permet à partir d'un tableau de notes et d'une note de référence de renvoyer un tableau de booléens qui contient le résultat de la comparaison entre la note du tableau et la note de référence : `public static boolean[] estPlusGrandEgale(double[] notes, double note)`. Sur notre exemple nous aurions pour estPlusGrandEgale(ue2,8) comme résultat le tableau : {true, false, true, true}.

3.6.4. semestreObtenu

Ecrire une méthode qui permet à partir de deux tableaux de notes de renvoyer un tableau de booléens qui contient le résultat du semestre : `public static boolean[] semetreObtenu(double[] ue1, double[] ue2)`. On suppose que ue1 et ue2 ont la même taille. Pour rappel, le semestre est obtenu si la moyenne est supérieure ou égale à 10 et si chaque UE est supérieure ou égale à 8. Sur notre exemple nous aurions pour semestreObtenu(ue1,ue2) comme résultat le tableau : {false, false, true, true}.

3.6.5. numéros

Il est préférable pour l'enseignant de manipuler des tableaux de numéros d'étudiant. Ecrire une méthode qui prend en paramètre un tableau de booléens et qui retourne un tableau d'entiers qui contient les numéros des cases dont le valeur est vraie : `public static int[] numeros(boolean[] t)`. Sur notre exemple nous aurions pour numeros(semestreObtenu(ue1,ue2)) comme résultat le tableau : {2, 3}.

3.6.6. nomsObtenusSemestre

Ecrire une méthode qui permet à partir des noms des étudiants, des notes des UE de renvoyer un tableau de nom d'étudiants ayant le semestre : `public static String[] nomsObtenusSemestre(String[] noms, double[] ue1, double[] ue2)`. On suppose que noms, ue1 et ue2 ont la même taille. Sur notre exemple nous aurions pour nomsObtenusSemestre(nom,ue1,ue2) comme résultat le tableau : {"KATAE", "ATLAIT"}.

3.7. Exercice 7 (facultatif)

Le but est de créer un jeu de morpion, composé d'un carré de 3x3. Créer une classe `Morpion` et une classe `JeuMorpion`, la deuxième classe contiendra le `main`.[3].

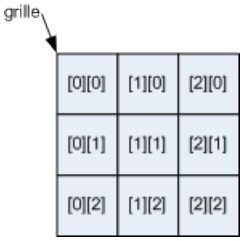
Notre grille de jeu est un tableau à deux dimensions.Les cases vides contiendront des 0, les cases du premier joueur contiendront 1 et celles du deuxième joueur contiendront 2. Nous utiliserons la convention grille[x][y], la case en haut à gauche est grille[0][0], la case en haut à doite est grille[2][0], la case en bas à droite est grille[2][2] et la case en bas à gauche est grille[0][2].

[2]En java, plusieurs méthodes peuvent avoir le même nom, si l'on peut les différencier par le nombre, l'ordre et le type de paramètres, l'on parle alors de surcharge.

Nous verrons plus tard en programmation que nous aurions pu avoir une seule méthode en utilisant le polymorphisme.

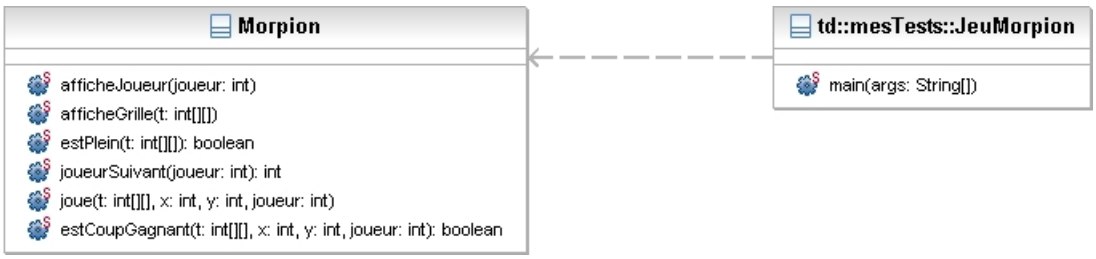
[3]Pour les tableaux à deux dimensions, vous pouvez regarder la documentation du devoir maison.

Figure 3.8. Grille de jeu



Les méthodes qui suivent, exception faite du main, sont des méthodes de classe de la classe `Morpion.java`.

Figure 3.9. Diagramme de classe de Morpion et JeuMorpion



3.7.1. Affichage du joueur

Coder la méthode `public static void afficheJoueur(int joueur)` qui pour joueur qui vaut 1 affiche : "Le joueur courant est :O." et qui pour joueur qui vaut 2 affiche "Le joueur courant est :X."

3.7.2. Affichage de la grille

Coder la méthode `public static void afficheGrille(int[][] t)` qui affiche la grille de jeu.

Intiallement nous aurons :

```
-------
| | | |
-------
| | | |
-------
| | | |
-------
```

Puis en cours de partie, nous pourrons avoir :

```
-------
|O| | |
-------
| | |X|
-------
| | | |
-------
```

3.7.3. Vérification si la grille est pleine

Coder la méthode `public static boolean estPlein(int[][] t)` qui renvoie `true` si la grille est plaine et `false` sinon.

3.7.4. Passer au joueur suivant

Coder la méthode `public static int joueurSuivant(int joueur)` qui renvoie 2 si le joueur est 1 et qui renvoie 1 si le joueur est 2.

3.7.5. La case est-elle jouable

Coder la méthode `public static boolean estJoubable(int[][] t, int x, int y)` qui renvoie `true` si la case est jouable et `false` sinon. Une case est jouable si elle est dans la grille et si elle est vide.

3.7.6. Jouer un coup

Coder la méthode `public static void joue(int[][] t, int x, int y, int joueur)` qui permet de jouer un coup. Cette méthode ne vérifie pas si la case est jouable.

3.7.7. Un coup est-il gagnant

Coder la méthode `public static boolean estCoupGagnant(int[][] t, int x, int y, int joueur)` qui permet de savoir si un coup est gagnant. Cette méthode ne joue pas le coup.

3.7.8. La partie (le main)

Ecrire le code du main de la classe `JeuMorpion.java` pour pouvoir jouer une partie. La capture qui suit est un exemple de partie possible.

```
-------
| | | |
-------
| | | |
-------
| | | |
-------
Le joueur courant est :O
coordonnées ?
x : 1
y : 1
-------
| | | |
-------
| |O| |
-------
| | | |
-------
Le joueur courant est :X
coordonnées ?
x : 1
y : 1
Injouable
x : 3
y : 3
Injouable
x : 0
y : 0
-------
|X| | |
-------
| |O| |
-------
| | | |
-------
Le joueur courant est :O
coordonnées ?
x : 0
y : 1
-------
|X| | |
-------
|O|O| |
-------
| | | |
-------
Le joueur courant est :X
```

```
coordonnées ?
x : 2
y : 2
-------
|X| | |
-------
|O|O| |
-------
| | |X|
-------
Le joueur courant est :O
coordonnées ?
x : 2
y : 1
-------
|X| | |
-------
|O|O|O|
-------
| | |X|
-------
Le joueur courant est :O
```

Chapitre 4. Travaux pratiques

Nous allons étudier en travaux pratiques les classes et les objets, puis les collections (listes et piles) et enfin nous reviendrons sur les classes et objets pour les intégrer avec des tableaux.

1. Classes et Objets

Les tableaux que nous avons vu sont des objets accessibles via une syntaxe particulière. Nous allons développer la généralité en Java.

1.1. Introduction aux objets en java

N'étant pas encore en programmation, nous allons présenter les objets comme des structures de données.

1.1.1. Définition

Nous ne prétendons pas ici refaire la théorie des langages à objets. Cependant nous allons préciser certaines notions. Une classe est une description statique d'une famille d'objets ayant même structure et même comportement. Les deux aspects sont donc :

- la donnée d'une composante structurelle (non dynamique) : *variables d'instances* (ou champs, ou attributs, ou propriétés) qui caractérisent l'état d'un objet pendant l'exécution d'un programme.
- la donnée d'une composante dynamique : procédures ou fonctions appelées *méthodes*. Les méthodes manipulent les variables d'instances.

La classe est donc un plan de construction, permettant d'obtenir des objets tous semblables (à l'instar d'un véhicule automobile d'une marque et d'un modèle particulier) mais tous différents (chaque véhicule possède sa couleur, son immatriculation,... le fait de repeindre un véhicule ne modifie pas la couleur des autres).

Un objet est une entité indépendante (dont la structure est connue de lui seul). Pour agir sur un objet, il faut utiliser les méthodes offertes par celui-ci. Cette utilisation passe par l'envoi d'un message (qui peut être vu comme une requête) à l'objet.

Dans la suite, nous allons définir et utiliser des objets .

1.1.2. Exemple : le point

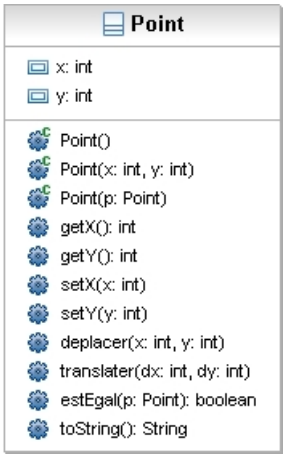
Nous allons d'abord travailler avec la classe Point, nous allons apprendre à l'utiliser puis à la créer.

Les points sont pour nous des points en deux dimensions caractérisés par leur abscisse et leur ordonnée. Bien que cela soit en dehors du domaine de ce cours nous illustrerons nos propos avec des diagramme UML. Vous reverrez par la suite de votre scolarité ces diagrammes.

1.1.2.1. Représentation UML (diagramme de classe)

La notation UML (Unified Modeling Language) et un moyen de décrire graphiquement des données et des traitements. Cette notation est composé d'un ensemble de diagramme, nous allons utiliser l'un d'entre eux qui permet de décrire une classe : le diagramme de classe

Figure 4.1. Diagramme de classe de Point



Sur ce diagramme nous voyons que notre classe se compose :

- un nom : `Point`
- de variables d'instance (ou propriétés, ou attributs ou champs) : *x* et *y*. L'état d'un point est défini par *x* et *y* qui sont deux entiers.
- d'un ensemble de méthodes qui vont définir le comportement d'un point :
  - Trois constructeurs qui vont permettre de construire des points. Ce sont les constructeurs qui sont invoqués lors de l'utilisation du mot clef *new*. Les constructeurs portent le noms de la classe et ne renvoient rien. Ici nous avons :
    - `Point()` qui construit un point en cordonnées (0,0), on parle de constructeur sans paramètre.
    - `Point(int, int)` qui construit un point a une abscisse et une ordonnée choisie.
    - `Point(Point)` qui fabrique un point à partit d'un autre point, on parle de constructeur par recopie, il a pour but d'éviter les alias.
  - Des méthodes qui vont permettre de se renseigner sur l'état et de modifier l'état de l'objet :
    - Les accesseurs (getters) qui permettent de connaître l'état :
      - `getX():int` qui retourne l'abscisse du point
      - `getY():int` qui retourne l'ordonnée du point
      - `estEgal(Point)` qui permet de comparer le point courant et un autre point .
    - Les manipulateurs (setters) qui permettent de modifier l'état de l'objet
      - `setX(int)` qui modifie l'abscisse
      - `setY(int)` qui modifie l'ordonnée
      - `deplacer(int,int)` et `translater(int,int)` qui modifient les coordonnées.

Nous avons une description de la classe `Point` nous en ferons le codage plus tard mais nous allons voir maintenant ~~comment créer de~~s objets points et les utiliser[1].

[1] En Java, la javadoc vous fournira une descriprion standard pour les classes.

1.1.2.2. Création et utilisation des objets

Les classes sont des types, pour déclarer une référence sur un objet nous utiliserons la syntaxe suivante :

```
NomClasse nomObjet;
```

.

Ce qui nous créé une référence (nomObjet) vers un objet de type NomClasse. Pour créer ou construire ou instancier l'objet, il faut faire appel à un constructeur en utilisant le mot clef *new*. :

```
nomObjet = new NomClasse([parametre])
```

Le constructeur défini l'état initial de l'objet, les variables d'instance sont crées et ont reçu une valeur, pour que l'objet évolue, ait un comportement, il faut utiliser des méthodes. L'accès aux variables d'instance et aux méthodes publiques de l'objet se fait en utilisant la notation . :

```
nomObjet.variablesInsatnce
nomObjet.nomMethode([parametre])
```

Nous pouvons maintenant appliquer ces concepts à l'utilisation de la classe Point.

Figure 4.2. Teste d'un point

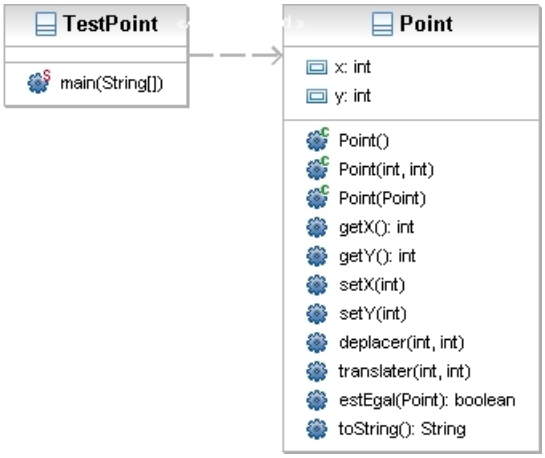
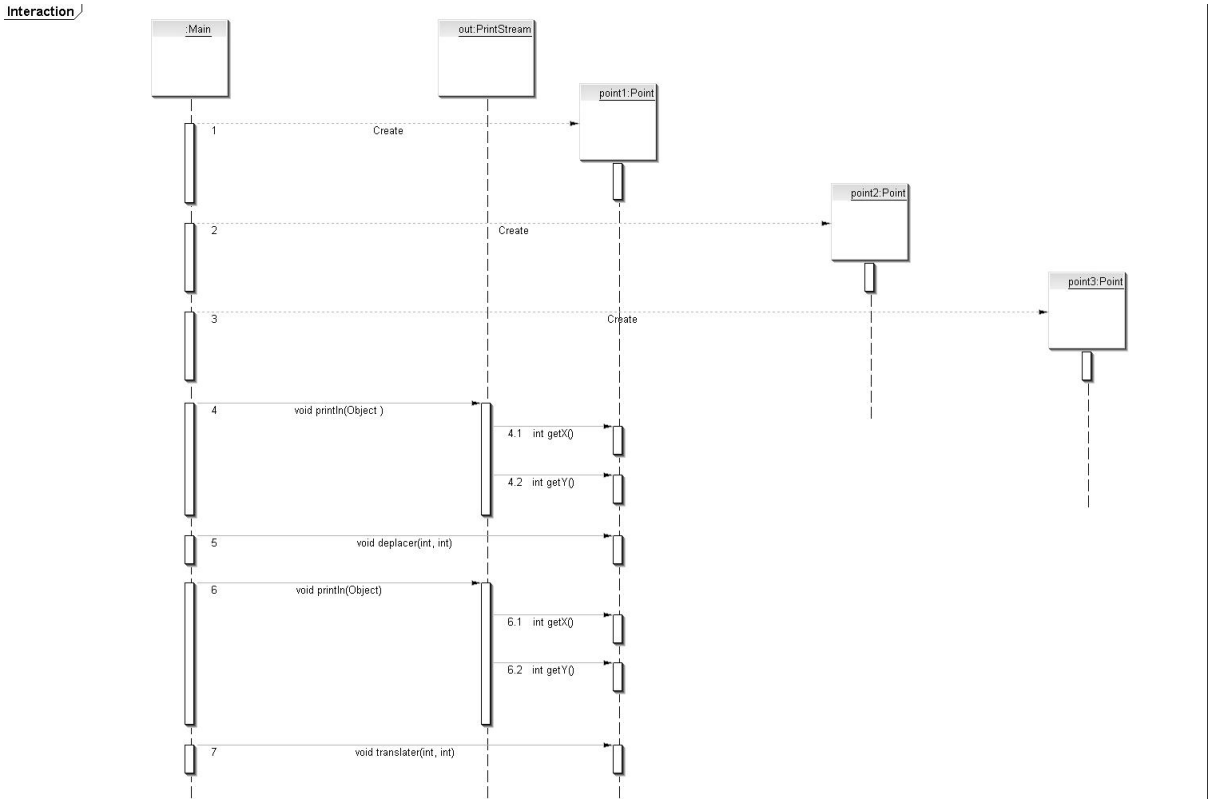


Figure 4.3. Diagramme des séquences



Exemple 4.1. Utilisation de la classe Point

```
public class TestPoint {

 public static void main(String[] args) {

  Point point1;              //point1 est une reference sur un objet de type Point
  Point point2;              //point2 est une reference sur un objet de type Point
  Point point3;              //point3 est une reference sur un objet de type Point

  point1 = new Point();      //le constructeur sans parametre qui créé
                             //un point en (0,0) est appelé.
                             //le point1 reference le Point cree on parle
                             //d'instanciation
  point2 = new Point(1,2);   //le constructeur prenant en parametre deux entiers
                             //en utilisé
                             //le point est en (1,2)
  point3 = new Point(point1);//le constructeur par recopie est utilisé, point3 est
                             //un clone de point2
                             //le point est en (0,0)

  System.out.println(point1.getX()+" " +point1.getY());
                             //getX() renvoie le x, getY() renvoie le y de Point
                             //0 0
  point1.deplacer(10, -5);   //point1 va en (1O,-5)
  System.out.println(point1.getX()+" " +point1.getY());
                             //10 -5
  point1.translater(2, 5);   // le x de point1 est augemente de 2
                             // le y de point1 est augemente de 5
  System.out.println(point1.getX()+" " +point1.getY());
                             //12 0
  point2.setY(10);           // le y de point2 prend 10
  System.out.println(point2.getY());
                             //10
  System.out.println(point2.getX()+" " +point2.getY());
                             //1 10
  System.out.println(point3.getX()+" " +point3.getY());
                             //0 0
  point3 = point2;           //point3 recoit la reference point2
                             //point3 est un alias sur point2
  point3.deplacer(5, 5);     //point3 est déplacé en (5,5)
  System.out.println(point2.getX()+" " +point2.getY());
                             //5 5
  System.out.println(point3.getX()+" " +point3.getY());
                             //5 5
 }
}
```

Bien évidement avant d'utiliser une classe, il faut l'avoir créée.

1.1.2.3. Codage de la classe Point

Nous allons maintenant étudier le codage de la classe Point, à vous de lire le code suivant et de poser des questions à votre enseignant(e) :

```
package tp.mesClasses;

/**
 * La classe Point définie des points caractérisés par une abscisse x et une ordonnée
 * @author jub
 *
 */
public class Point {
 /**
  * un point est caractèrisé par son abscisse x et son ordonnée y
  */
 private int x, y;
```

```
/**
 * Construit un Point en coordonnées (0,0)
 */
public Point() {
 //this(0,0);
 this.x = 0;
 this.y = 0;
}

/**
 * Construit un Point de coodonnées (x,y) choisies
 * @param x l'abscisse choisie
 * @param y l'ordonnée choisie
 */
public Point(int x, int y) {

 this.x = x;
 this.y = y;
}

/**
 * Construit un point qui est la copie d'un autre
 * @param p l'autre Point
 */
public Point(Point p) {
 //this(p.x,p.y);
 this.x = p.x;
 this.y = p.y;
}
/**
 * Renvoie l'abscisse
 * @return l'abscisse
 */
public int getX() {
 return this.x;
}
/**
 * Renvoie l'ordonnée
 * @return l'ordonnée
 */
public int getY() {
 return this.y;
}
/**
 * Permet de modifier l'abscisse
 * @param x la nouvelle abscisse
 */
public void setX(int x) {
 this.x = x;
}
/**
 * Permet de modifier
 * @param y
 */
public void setY(int y) {
 this.y = y;
}

/**
 * Deplace le point aux nouvelles coordonnées
 * @param x la nouvelle abscisse
 * @param y la nouvelle ordonnée
 */
public void deplacer(int x, int y)
{
 this.x = x;
 this.y = y;
}

/**
```

```
  * Translate le point de dx en abscisse et dy en ordonnée
  * @param dx la translation en abscisse
  * @param dy la translation en ordonnée
  */
 public void translater(int dx, int dy)
 {
  //this.x +=dx;
  //this.y +=dy;
  this.x = this.x + dx;
  this.y = this.y + dy;
 }
 /**
  * Permet la compraison entre le point courant et un autre point
  * @param p l'autre point
  * @return true si les abscisses et les ordonnées sont égales et false sinon.
  */
 public boolean estEgal(Point p)
 {
  return (this.x == p.x && this.y == p.y);
 }

 /* (non-Javadoc)
  * @see java.lang.Object#toString()
  */
 public String toString() {
  return "Point [x=" + this.x + ", y=" + this.y + "]";
 }

}
```

Vous pouvez maintenant créer un paquetage tp.mesClasses dans lequel vous coderez la classe `Point` et un paquetage tp.mesTests dans lequel vous coderez la classe `TestPoint`.

Le diagramme de classe UML ne fournit que le nommage, pour coder, il nous faut de la documentation. Javadoc est le standard industriel pour la documentation des classes Java. Vous allez suivre la procedure suivante pour générer la JavaDoc de la classe Point :

1. Dans eclipse choisir project
2. Puis choisir generate JavaDoc
3. Indiquer l'emplacement de javadoc.exe (C:\Program Files (x86)\Java\jdk1.6.0_21\bin\javadoc.exe chez moi)
4. Choisir private
5. Generer
6. Observer les variables d'instances, constructeurs et méthodes.

1.1.3. Utilisation d'une API

Une API (Application Programming Interface) ou interface de programmation pour les applications comprend toutes les méthodes et les variables utilisables pour les programmeurs pour écrire leurs applications. L'API d'une classe décrit un objet et la manière de le manipuler.

1.1.3.1. Principes d'utilisation

L'Application Programme Interface standard est l'ensemble des bibliothèques mises à votre disposition en Java. Les APIs sont disponibles sur `http://docs.pedago.src/java/api/`ou sur `http://java.sun.com/javase/6/docs/api/`.

Rappels :

- la première étape est de déclarer une instance d'objet,

- avant d'utiliser une méthode sur une instance d'objet, il FAUT construire cette instance (par une clause *new*),
- pour utiliser une méthode d'une classe, il faut l'appliquer à une instance d'un objet de cette classe. Exemple : maString.methodeDeString(paras),
- faire attention aux types des paramètres et au type de retour de la méthode,
- bref, le principe c'est de respecter les signatures des méthodes et de suivre la procédure :

  1. déclaration
  2. construction
  3. utilisation

  des objets.

1.1.3.2. Exemple de l'API d'une classe *Bidon*

Une classe *Bidon.java* a été réalisée, comprenant des constructeurs d'objets et des méthodes s'appliquant sur ces objets :

```
public class Bidon ...
// Constructeurs
public Bidon(int a, int b);
```

Le constructeur, du nom de la classe, prend en paramètre deux entiers.

```
//méthodes
public int bidule(String c);
```

renvoie un entier comme résultat, prend une *String* en paramètre.

1.1.3.3. Exemple d'utilisation de la classe *Bidon*

On peut vouloir utiliser la classe `Bindon.java` en instanciant des objets de cette classe et en appliquant certaines méthodes sur ces objets. On va pour cela, créer une classe `TestBidon.java` comprenant :

déclaration
: de ma variable *monObjet* de type *Bidon*

```
Bidon monObjet ;
```

construction
: en donnant des valeurs aux deux entiers requis par le *constructeur*

```
monObjet = new Bidon(2,3) ;
```

appel
: de la méthode *bidule* ; on stocke le résultat dans une variable de type *int*, on donne une chaîne de caractères en paramètre.

```
int entier ;
entier = monObjet.bidule("bonjour") ;
```

1.1.3.4. Utilisation de l'API exmple String et StringBuffer

Avant de coder nos propres classes, nous allons continuer à nous familiariser avec les objets en en créant et en en manipulant.

1.1.3.4.1. Principe

L'objet *String* décrit une chaîne de caractères, non modifiable, de taille fixe. En d'autres termes, dès qu'une *String* est créé, elle devient non modifiable. La *String* est le seul objet à pourvoir être créé sans appel explicite du constructeur :

```
String s = "bonjour" ;
```

Ceci est dû au caractère incontournable des objets chaînes de caractère. L'objet *StringBuffer* décrit une chaîne de caractères, modifiable, de taille variable, ce qui permet d'insérer des caractères à une position, de rajouter des caractères en fin, ...

Ces deux classes sont deux classes de la bibliothèque standard de Java. La documentation de ces classes se trouve donc dans l'API standard (en anglais).

1.1.3.4.2. Réalisation

Vous allez :

- Créer une classe TestString, que vous utiliserez pour la suite.
- Créer une String initialisée à "anticonstitutionnellement".
- Quelle est la longueur de cette String ? (faire afficher le résultat).
- Afficher le troisième caractère.
- Extraire la sous-chaîne allant du deuxième caractère au quatrième, l'afficher.
- Passer la String du départ en majuscule.
- Faire écrire ce résultat et la String originale.
- Afficher la première et la dernière position du caractère 'n' dans la String. Que se passe-t-il, si on demande la position d'un caractère non présent dans la chaîne ?
- Créer un nouvelle String initialisée à bonjour.
- Créer une StringBuffer à partir de la String précédente.
- Concaténer " le monde", faire afficher la StringBuffer résultat.
- Insérer " tout" à la position 7, faire afficher la StringBuffer résultat.
- Créer trois *String s1*, *s2* et *s3* en appelant explicitement le constructeur (

  ```
  String s = new String(...)
  ```

). Les trois valeurs d'initialisation sont *"bonjour"*, *"bonjour"*, et *"Bonjour"*. Comparer *s1* à *s2* et *s1* à *s3* en utilisant successivement l'opérateur == , la méthode *equals(...)* et la méthode *equalsIgnoreCase(...)*. Résultat ?

1.1.3.5. Utilisation de l'API java.util.Date

La classe `java.util.Date` n'est pas d'un usage cohérent, je vous conseil de lire la documentation. Vous observerez aussi qu'elle contient des méthodes dépréciées. Une méthode dépréciée est une méthode dont-on conseil l'utilisation d'une autre méthode plus récente. Ici nous utiliserons les méthodes dépréciées.

1.1.3.5.1. Introduction

La classe *Date* permet de manipuler des dates. Cette classe fait partie du paquetage java.util. Il faut donc mettre

```
import java.util.Date;
```

en première ligne d'un fichier qui veut utiliser cette classe. Certaines méthodes de cette classe sont dépréciés, mais nous allons tout de même l'utiliser. Sa remplaçante présente (pédagogiquement parlé) beaucoup moins d'intérêts. Il ne faut donc pas s'inquiéter de messages comme :

```
Note : TestDate.java uses or overrides a deprecated API.
```

```
Note : Recompile with -Xlint :deprecation for details.
```

1.1.3.5.2. Réalisation

Vous aller :

- Créer une date initialisée à la date du jour
- L'afficher
- Afficher le mois,
- Modifier l'année en 1999, afficher la date
- Créer une date initialisée à votre date de naissance, l'afficher
- Comparer ces deux dates.

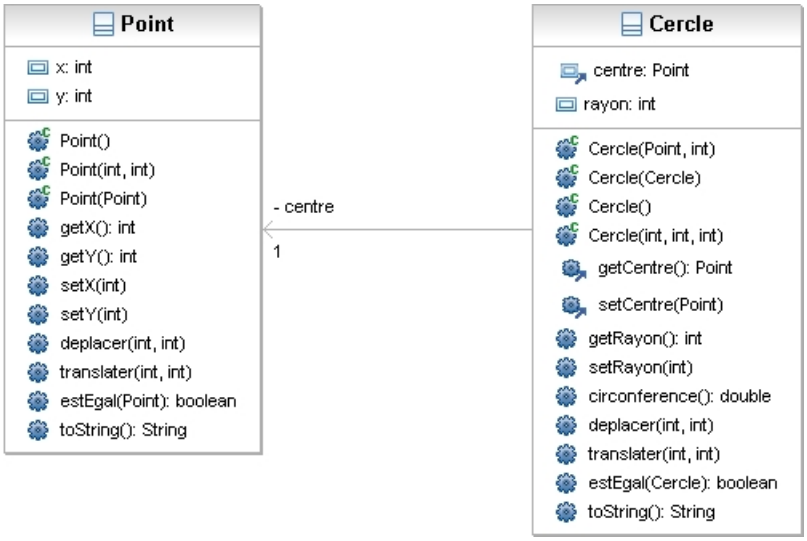
1.1.4. Codage de la classe Vehicule et de la classe TestVehicule

Un véhicule se démarre, s'accélère, se freine, s'arrête, ... On peut consulter la valeur de la vitesse (comme sur un compteur de voiture). On peut comparer la vitesse de deux véhicules. Il est préférable de n'actionner le démarreur que si la voiture n'est pas en marche... Proposer un diagramme de classe UML, puis une implémentation reflétant la situation[2].

1.1.5. Codage de la classe Cercle et de la classe TestCercle

Le cercle est pour nous un point augmenté d'un rayon.

Figure 4.4. Association entre Point et Cercle



Le premier constructeur Cercle() permet de créer un cercle de rayon 1 et de centre (0,0). Le deuxième constructeur Cercle(Cercle) permet de construire un cercle à partir d'un autre cercle passé en paramètre. Le troisième constructeur Cercle(int, int, int) permet de créer un cercle à partir d'une abscisse, d'une ordonnée et d'un rayon. Le quatrième constructeur Cercle(Point, int) permet de créer un cercle à partir d'un point et d'un rayon.

Avant de proposer une implémentation de la classe cercle, vous aller importer sa JavaDoc :

1. Télécharger `doc-cercle.zip`
2. Clique droit sur le projet

[2]Dans le diagramme de classe vous ne pourrez exprimer : Il est préférable de n'actionner le démarreur que si la voiture n'est pas en marche ...

3. import -> general -> archive file

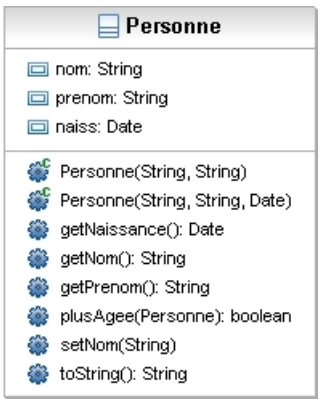
Suivre la JavaDoc pour proposer une implémentation de la classe Cercle, puis la tester en suivant la procèdure suivante :

1. Créer un classe TestCercle qui
2. contient un main
3. qui lui même va créer quatre cercles :
    1. cercle1 en (0,0) avec un rayon de 1 créé avec le premier constructeur
    2. cercle2 un cercle créé à partir du premier cercle
    3. cercle3 un cercle avec comme abscisse 1, ordonnée 1 et rayon 10
    4. cercle4 un cercle construit à partir d'un point de cordonnées (2,2) avec un rayon de 20
4. Une fois les objets construits, afficher la circonférence de cercle1.
5. Déplacer cercle2 en (2,2).
6. Faire un test permettant de savoir si cercle3 et cercle4 sont égaux.

1.1.6. Codage de la classe Personne et de la classe TestPersonne

Tout comme ma classe Cercle importer la JavaDoc de la classe Personne (`doc-personne.zip`), puis coder la classe et son test.

Figure 4.5. Diagramme de classe de la classe Personne



2. Structures de données séquentielles

Nous commencerons par une approche itérative avec la liste et la pile. Pour ceux qui le souhaitent une approche récursive est proposée en option.

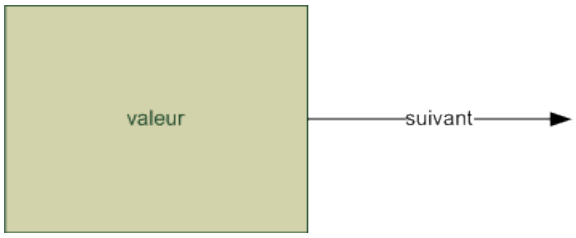
2.1. Cellule

Nous allons commencer par définir une cellule. Les cellules nous permettrons plus tard de créer des listes simplement chaînées.

2.1.1. Présentation

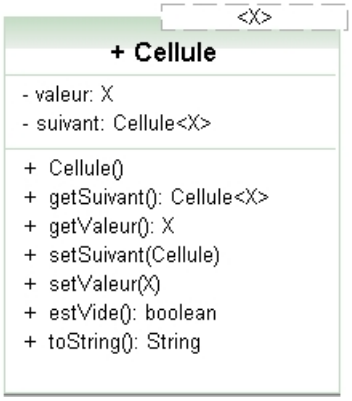
Une cellule est définie par une valeur et une référence vers une autre cellule.

Figure 4.6. Cellule



Une référence qui n'a pas encore reçu de valeur a la valeur *null*.

Figure 4.7. Diagramme de classe de Cellule



Le X que vous observé est un type passé en paramètre, ainsi la classe Cellule est indépendante du type, on parle de généricité. Sans la généricité nous aurions soit utiliser le polymorphisme, soit créer un classe par type, comme par exemple `CelluleEntier`, `CellulePersonne`, ...

2.1.2. Codage

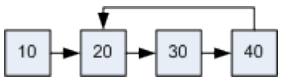
Donner le code la classe `Cellule` sachant que le constructeur construit une cellule vide. Vous pouvez vous inspirer du code suivant et utiliser les assistants d'eclipse.

```
public class Cellule<X> {
 private X valeur;
 private Cellule<X> suivant;
}
```

2.1.3. Première utilisation de la cellule

Dans une classe TestCellule reproduire le schéma suivant, il vous faudra utiliser le debugueur pour vérifier. Le type objet correspondant aux int est `Integer`.

Figure 4.8. Utilisation de cellules

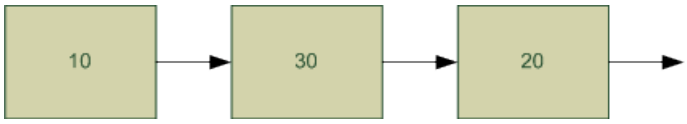


2.2. Liste simplement chaînée en version itérative

Le principe de la liste chaînée est que chaque élément possède, en plus de la donnée, une ou des références vers les éléments qui lui sont logiquement adjacents dans la liste.

Une liste simplement chaînée est une liste qui ne possède qu'une référence, pour nous elle sera vers la cellule de droite.

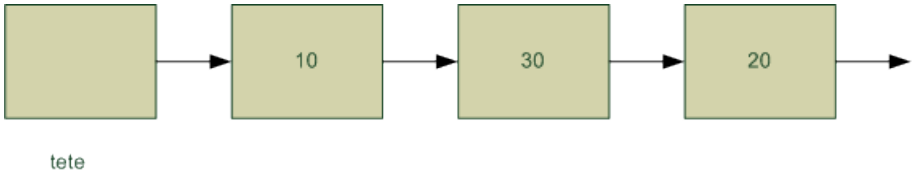
Figure 4.9. Liste simplement chaînée



2.2.1. Présentation

Nous souhaitons que notre classe Liste soit accessible via une cellule particulière nommée "tete". Cette cellule ne contient pas de valeur mais permet simplement de connaître le début de la liste donc la liste entière.

Figure 4.10. Liste simplement chaînée réelle



2.2.2. Codage

Vous aller réaliser l'implémentation d'une liste répondant au diagramme de classe UML suivant :

Figure 4.11. Diagramme UML de la classe Liste

Donner pour chacune des étapes qui vont suivre le code java correspondant et l'ordre de grandeur de la complexité temporelle.*L'ordre des méthodes est pédagogique, dans la réalité, il faudrait commencer par les méthodes impliquant la position, les autres pouvant être déduites de ces dernières.*

2.2.2.1. Variable(s) d'instance(s)

La ou les variables d'instance. Justifier de ces ou de cette variable.

2.2.2.2. Constructeur sans paramètre

Donner le code java correspondant.

2.2.2.3. Méthode estVide

Une méthode pour savoir si la liste est vide.

2.2.2.4. Méthode getPremier

Accès au premier élément de la liste.

2.2.2.5. Méthode getDernier

Accès au dernier élément de la liste.

2.2.2.6. Méthode insereTete

On insère une nouvelle valeur en tête de liste.

2.2.2.7. Méthode insereQueue

On insère une nouvelle valeur en queue de liste.

2.2.2.8. Méthode suppressionTete

On supprime la valeur en tête de liste.

2.2.2.9. Méthode suppressionQueue

On supprime la valeur en queue de liste

2.2.2.10. Exercices complémentaires

Nous pouvons enrichir notre liste avec des méthodes, pour pouvoir l'utiliser comme un tableau de taille variable.

2.2.2.10.1. Méthode getTaille

Renvoie le nombre d'éléments de la liste que nous recomptons à chaque fois.

> **Note**
>
> Il existe bien entendu, une meilleures solution qui consiste à rajouter une variable d'instance incrémentée à chaque insertion et décrémentée à chaque suppression dans la liste.

2.2.2.10.2. Méthode getVal

Renvoie l'élément à une position quelconque.

2.2.2.10.3. Méthode setVal

Modifie un élément à une position quelconque.

2.2.2.10.4. Méthode insertPosition

Insère un élément à une position quelconque.

2.2.2.10.5. Méthode suppressionPosition

Supprime l'élément à une position quelconque.

2.3. Pile

Une pile est une structure de données de type LIFO (Last In First Out). Ce qui signifie que l'élément inséré en dernier dans une pile est le premier à en être extrait. L'analogie avec une pile d'assiette nous amène facilement à définir les méthodes de la Pile.

2.3.1. Présentation

Nous allons utiliser une liste pour implémenter notre Pile.

La pile possède trois méthodes :

1. estVide qui permet de savoir si la pile est vide.
2. push qui permet de rajouter un élément
3. pop qui permet de retire un élément

Figure 4.12. Diagramme de classe de Pile

2.3.2. Codage

Donner le code de la classe Pile.

3. Structures de données approche récursive (facultatif)

Une méthode est récursive si elle s'appelle elle même. La récursivité permet d'appréhender de nouvelles structures comme les arbres ou les graphes.

3.1. Rappel sur la récursivité

Il est des problèmes en algorithmique comme les parcours d'arbres qui ne peuvent être résolus simplement sans récursivité. La récursivité consiste a s'appeler soit même. Le principe se retrouve en art, en linguistique et bien évidement en informatique.

3.1.1. Définition

On appelle récursivité le fait pour un sous-programme (méthode) de s'appeler au moins une fois. La vision récursive s'oppose bien souvent à la vision itérative.

3.1.2. Exemples

Pour réaliser une méthode récursive, il faut :

- un point d'arrêt
- trouver un sous problème identique au problème et exprimer le résultat en fonction de ce sous problème.

3.1.2.1. Factorielle

Le calcul d'une factorielle peut s'exprimer de deux manières :

- n! = n * n - 1 * ... * 1
- n! = n * (n -1)!

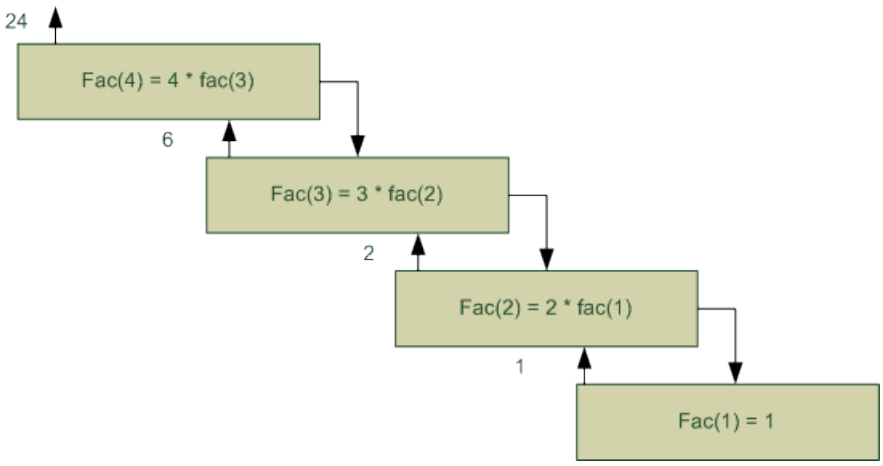
La première version permet l'implantation d'une fonction itérative. La deuxième permet une implantation récursive de la fonction factorielle.

Le cas d'arrêt et 0! = 1, le sous problème (n-1)!, et le moyen de relier le problème n! au sous problème.

Exemple 4.2. Factorielle récursive

```
public static int facR(int n)
 {
  int res;
  if (n==0 || n==1) //Cas d'arret
  {
   res = 1;
  }
  else
  {
   res = n * facR(n-1); //Appel recursif
  }
  return res;
 }
```

Figure 4.13. Appel récursifs de la factorielle



3.1.2.2. Somme d'un tableau d'entiers

Le cas d'arrêt est le tableau à une case, nous pouvons conclure que sa somme est celle de sa case. L'appel récursifs est obtenu en constatant que la somme des éléments est égale au premier élément plus la somme du sous tableau restant. Nous supposons la méthode extraction qui permet d'extraire un sous tableau est donnée

Important

Ce premier exemple est a oublier aussitôt car a chaque appel, il y a recopie du tableau, une version plus performante suivra.

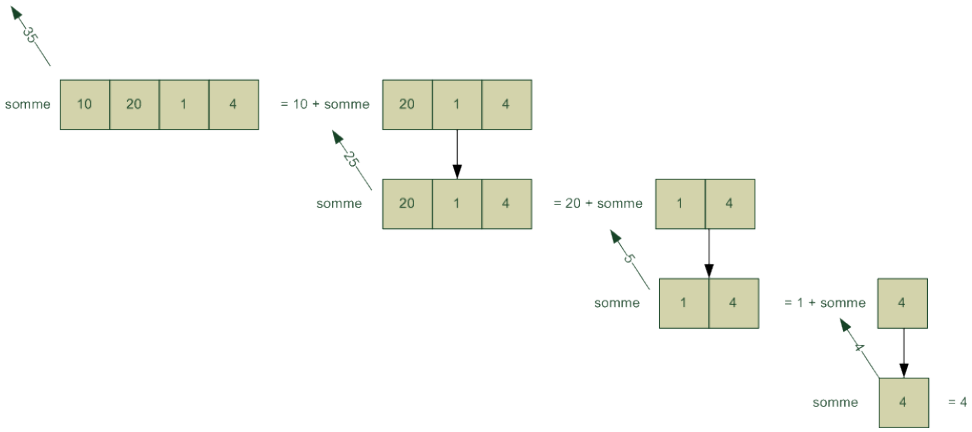
:

```
public static int[] extraction(int[] t, int debut, int fin)
 {
  int[] res;
  if (0<fin-debut && fin-debut < t.length)
  {
   res = new int[fin-debut+1];
   for (int i=0; i < res.length; i++)
   {
    res[i]=t[debut+i];
   }
  }
  else
  {
   res = null;
  }
  return res;
 }
```

Exemple 4.3. Somme récursive des éléments d'un tableau d'entiers

```
public static int sommeR(int[] t)
 {
  int res = 0;
  if (t.length ==1)
  {
   res = t[0];
  }
  else
  {
   res = t[0]+sommeR(extraction(t, 1, t.length-1));
  }
  return res;
 }
```

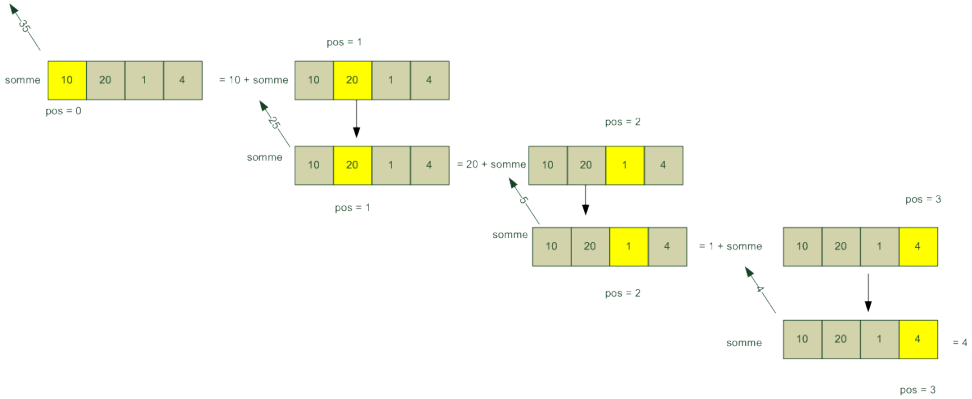
Figure 4.14. Appel récursif de la somme des éléments d'un tableau d'entiers



La solution précédente consomme de la mémoire, à chaque appel récursif un tableau est créé, nous pouvons proposer une nouvelle version avec un marqueur qui indique la position du sous-tableau à traiter :

```
public static int sommeR2(int[] t, int pos)
 {
  int res;
  if (pos == t.length -1)
  {
   res = t[pos];
  }
  else
  {
   res = t[pos] + sommeR2(t, pos +1);
  }
  return res;
 }
```

Figure 4.15. Appel récursif de la somme des éléments d'un tableau d'entiers avec un index.



3.2. Liste simplement chaînée en version récursive

Nous allons reprendre la liste simplement chaînée pour en proposer cette fois-ci une version récursive.

Voici trois méthodes pour commencer la récursivité :

```
private X car()
 {
  X res = null;
  if (!estVide())
   res = this.tete.getSuivant().getValeur();
  return res;
 }

 private List<X> cdr()
 {
  List<X> res = new List<X>();
  if (!estVide())
   res.tete = tete.getSuivant();
  return res;
 }

 public boolean estVide()
 {
  return this.tete.getSuivant() == null;
 }
```

`car(x)` renvoie le premier élément de la liste, `cdr()` renvoie la liste privée du premier élément.

Figure 4.16. Liste récursive

Coder dans l'ordre suivant les méthodes suivantes.

3.2.1. getTaille

`public int getTaille()` renvoie la taille de la liste.

3.2.2. getVal

`public X getVal(int pos)` renvoie la valeur en position *pos*.

3.2.3. setVal

`public void setVal(X val,int pos)` modifie la valeur de l'élément de position *pos*.

3.2.4. insertionPosition

`public void insertionPosition(X val, int pos)` insert *val* en position *pos*.

3.2.5. suppressionPosition

`public void suppressionPosition(int pos)` supprime l'élément de position *pos*.

3.3. Arbres binaires

Nous allons implémenter un arbre binaire, à savoir un arbre dont chaque noeud possède au plus un fils gauche et un fils droit.

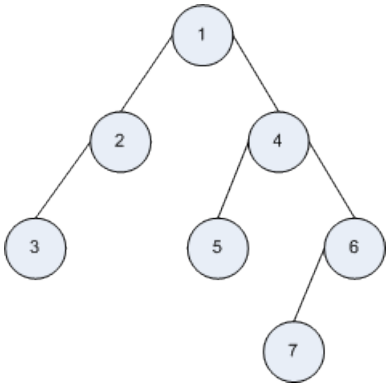
Figure 4.17. Classe ArbreB

Notre arbre binaire ne peut-être construit vide. Pour les exercices qui suivent, il vous faudra à chaque fois, au plus tôt tester vos méthodes.

3.3.1. Création de la classe

Créer la classe arbre binaire avec pour le moment, les attributs et les constructeurs, puis dans un fichier de test séparer reproduire l'arbre suivant :

Figure 4.18. Exercice arbre binaire



3.3.2. Les feuilles et les noeuds

Un noeud est un élément de l'arbre, un noeud particulier est la racine, les noeuds qui n'ont pas de fils sont appelés feuilles.

Coder la méthode `public boolean estFeuille()` qui renvoie `true` si l'arbre est une feuille et `false` sinon, coder de même la méthode `public boolean estNoeud()`.

3.3.3. La taille

La taille est le nombre de noeuds de notre arbre, coder : public int taille().

Ici, vous devez trouver sept.

3.3.4. La hauteur

La profondeur d'un noeud, est le est la longueur du chemin allant de ce noeud à la racine.

Ici la profondeur du noeud contenant 3 est 2.

La hauteur est la profondeur maximum, ici 3.

Coder la méthode `public int hauteur()`.

3.3.5. Parcours en préordre

Le parcours préordre est un parcours d'arbre qui repose sur la méthode suivante :

- examiner la racine
- parcourir en préordre le sous-arbre gauche
- parcourir en préordre le sous-arbre droite

Coder la méthode `public String parcoursPreordre()`.

Vous devez trouver : 1234567.

3.3.6. Parcours en postordre

Le parcours postodre est un parcours d'arbre qui repose sur la méthode suivante :

- parcourir en postordre le sous-arbre gauche
- parcourir en postordre le sous-arbre droite
- examiner la racine

Coder la méthode `public String parcoursPostordre()`.

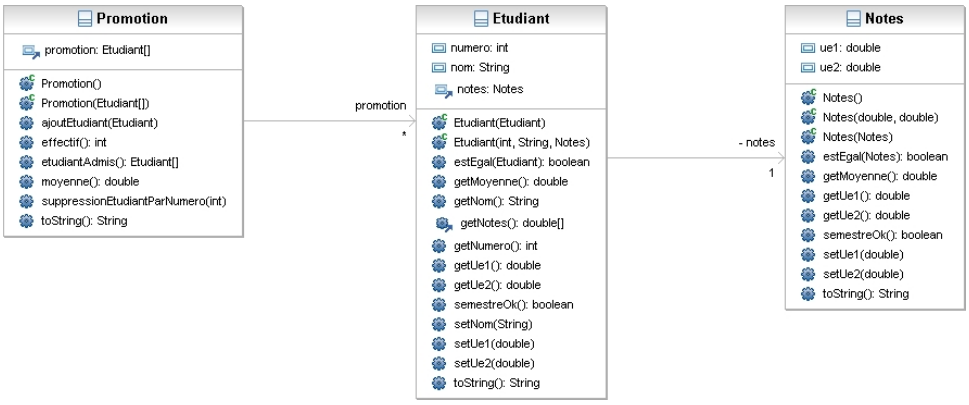
Vous devez trouver : 3257641.

4. Tableaux et objets

Nous avons appris d'un coté à utiliser des tableaux et d'un autre coté à créer des classes et des objets, nous allons maintenant les utiliser conjointement.

Vous aller coder en suivant la java dos les trois classes suivantes.

Figure 4.19. Une promotion d'étudiants qui a des notes



4.1. Étudiants et notes

Les notes représentent les notes de l'ue1 et de l'ue2, un étudiant a des notes, un numéro et un nom. Coder et tester `Notes` puis `Etudiant`.

4.2. Promotion

Pour la promotion, je vous propose deux étapes.

4.2.1. Avec une promotion non vide

Dans cette première version, vous ne coderez pas le constructeur sans paramètre ainsi que la méthode d'ajout et de suppression.

4.2.2. Avec une promotion vide

Cette fois-ci, vous allez rajouter le constructeur sans paramètre et les méthodes d'ajout et de suppression.*Les tableaux étant de taille fixe, vous devez pour ajouter ou supprimer un étudiant de sa promotion, créer un nouveau tableau avec respectivement un étudiant de plus ou un étudiant de moins puis affecter à la variable d'instance promotion ce nouveau tableau.*

5. Variables de classe

Le but de ce *TP* est de consolider l'écriture de classes décrivant un objet et d'y rajouter les variables et méthodes de classes.

5.1. Rappels

Les objets ont des comportements et des états séparés, pour qu'ils puissent partager des données, une solution est l'utilisation de variables de classes. Les variables de classes sont des variables partagées par tous les objets de cette classe. Les méthodes de classe permettent d'accéder à des méthodes sans avoir à créer d'objets. La classe utilitaire *java.lang.Math* en est un exemple, elle contient des constantes (

```
Math.PI
```

,

```
Math.E
```

) et des méthodes comme

```
double Math.sqrt(double x)
```

qui calcul une racine carré.

5.1.1. Variables de classes, constantes

Exemple 4.4. Variable de classe

```
public class MaClasse
{
static int maVariableDeClasse ;
...
```

Une variable de classe se déclare comme une variable d'instance, mais en rajoutant le mot clef *static*.

Une constante se déclare comme une variable de classe mais en rajoutant le mot clef *final*. La signification de *final* est : non modifiable, donc constant.

Exemple 4.5. Variable de classe constante

```
public class MaClasse
{
final static int MA_CONSTANTE ;
...
```

il est de tradition de noter les constantes en majuscules en séparant les (éventuels) mots composant l'identificateur par le caractère _.

Les variable de classe ou constantes peuvent s'initialiser de deux manière :

directement
: au moment de la déclaration.

Exemple 4.6. Initialisation et déclaration d'une variable de classe constante

```
public class MaClasse
{
static int maVariableDeClasse = 0 ;
...
```

dans un bloc d'initialisation static.
: C'est simplement un bloc délimité par static{ et } dans lequel on fait ces initialisations (traditionnellement, on utilise un tel bloc pour des initialisations plus complexes que des simples affectation).

Exemple 4.7. Bloc d'initialisation static

```
public class MaClasse
{
static String maVariableDeClasse ;
static
{
uneMethodeQuiInitialiseMaVariable(maVariableDeClasse) ;
}
...
```

Si on ne fait aucune initialisation, une initialisation par défaut (0 pour les types *byte*, *short*, *int*, *long*, *float*, *double*, false pour *boolean*, null pour un type *objet*) est faite.

Les variable de classe ou constantes peuvent se modifier n'importe quand (dans un constructeur, dans une méthode, ...).

5.1.2. Méthodes de classes

Une méthode de classe se définit comme une méthode d'instance, mais en rajoutant le mot clef static.

Une méthode de classe est une méthode qui n'est pas associé à une instance particulière. L'utilisation des méthodes de classes se fait pour une des raisons suivante :

- aucun objet n'est impliqué ; c'est la cas des méthodes de la classe Math ou des fonctions ou procédures que nous avons écrite en algorithme.
- on veut symétriser l'écriture d'une méthode opérant sur deux instances d'une classe (typiquement une méthode de comparaison) :

```
public static boolean soldeSuperieur(CompteBancaire compte1,
CompteBancaire compte2)
```

- on a une méthode qui est spécialisée dans la manipulation des variables de classes.

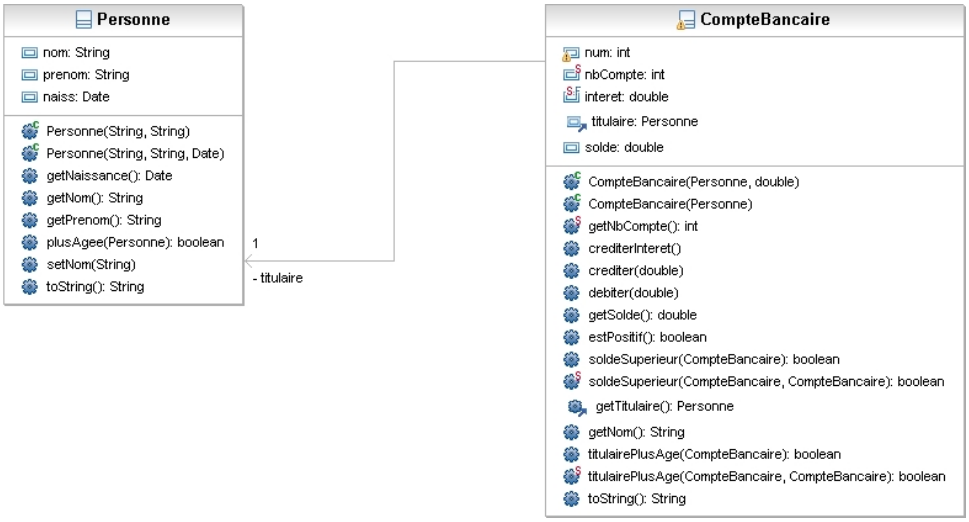
5.2. Classe CompteBancaire

Avant de commencer cette partie vous devez avoir la classe Personne opérationnelle, nous allons la réutiliser.

5.2.1. Introduction

La classe CompteBancaire décrit (sommairement) un compte bancaire. On mémorise deux informations, le titulaire (une Personne) du compte, le solde du compte. Ce compte peut être crédité et débité. On veut pouvoir gérer le nombre de comptes créés. On utilise une constante pour le taux d'intérêts, commun à tous les comptes et de (2,5% par exemple). Enfin, deux méthodes de classes permettent d'obtenir le nombre de comptes créés et de comparer le solde de deux comptes. Une modélisation UML de cette classe est donnée.

Figure 4.20. Comptebancaire



5.2.2. Implantation de la classe CompteBancaire

En utilisant l'API créer une classe *CompteBancaire* dans le paquetage *tp.etudiant.classes*.

5.2.3. Réalisation d'une classe test

Écrire une classe *TestCompte*, dans le paquetage *test*, dotée d'une méthode *main* qui fait appel aux différentes méthodes que vous aurez écrites dans la classe *CompteBancaire*.

6. Collections

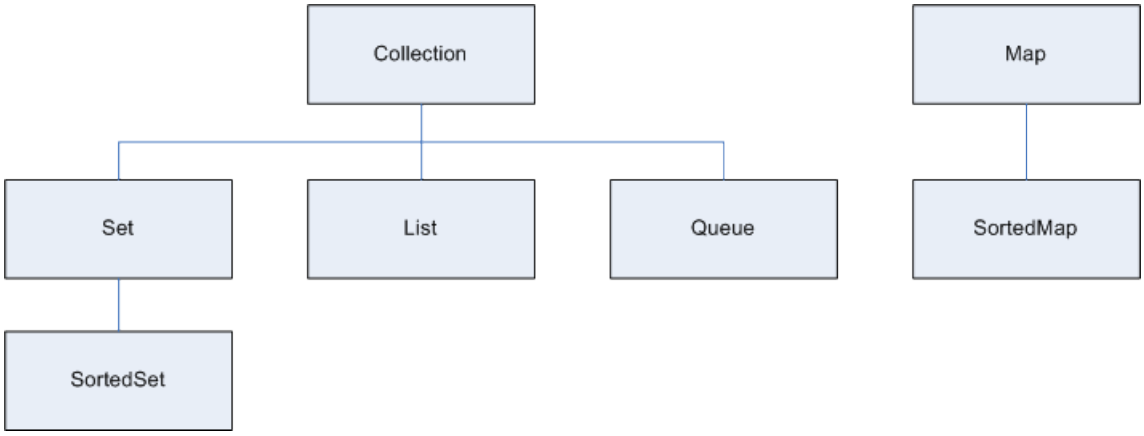
Les tableaux ne sont pas les seules structures de données utilisables en java, il y a aussi les *collections* et les "*maps*" (tableaux associatifs: clef-valeur).

Sur http://fmora.developpez.com/ vous trouverez le tutoriel dont cette partie est issue.

6.1. Présentation

Une collection est un groupe d'objets connus par ses éléments. Les collections peuvent ou non admettre des doubles, être triés, … Toutes les collections implémentes l'interface[3] « Collection » et s'utilisent « de la même manière ». Le nom des méthodes est fixée.

Figure 4.21. Collections



Pour comparer les collections l'analogie mathématique reste valide :

- Set des ensembles non ordonnés (pas de doublons),
- SortedSet des ensembles ordonnés,
- List qui représente une séquence,
- Queue qui représente une file,
- Map qui est une association clef valeur,
- SortedMap une Map ordonnées qui maintient un ordre croissant et est utilisée pour les dictionnaires, ...

Une collection peut être parcourues avec la construction *for-each* (une utilisation particulière de la structure *for*) ou avec un *itérateur*.

```
//Affiche tous les elements de la collection
for (Object o : collection)
System.out.println(o);
```

ou

```
for (Iterator<?> it = collection.iterator(); it.hasNext();
 System.out.println(it.next());
```

permet d'afficher tous les éléments d'une collection.

Un itérateur possède trois méthodes :

boolean hasNext();
vrai si il reste un élément,

[3]Nous verrons bientôt en programmation qu'une interface est un ensemble de déclarations de méthodes sans implantation qui doivent être implémentée avant d'être utilisée. Ainsi toutes les classes qui implémentent une même interface disposent des mêmes méthodes.

E next();
retourne l'élément courant et se positionne sur le suivant,

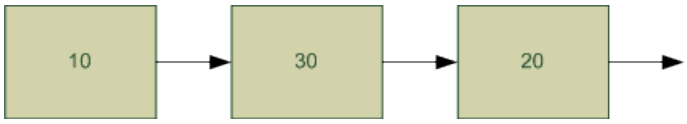
void remove();
retire le dernier élément accédé.

Les collections possède enfin l'avantage d'avoir des opérations qui portent sur l'ensemble des éléments d'un collection comme : *clear*, *removeAll*, *addAll*, …

6.2. Listes

Une liste est une collection qui permet un accès par position, qui peut être réarrangée, dans laquelle on peut faire des recherches.

Figure 4.22. Liste



List est une interface et ne peut être utilisée directement, il faut soit l'implémenter soit utiliser une classe qui implémente l'interface *List* (*AbstractList*, *AbstractSequentialList*, *ArrayList*, *AttributeList*, *CopyOnWriteArrayList*, *LinkedList*, *RoleList*, *RoleUnresolvedList*, *Stack*, *Vector*).

Exemple 4.8. Utilisation d'un vecteur

Cet exemple illustre l'utilisation d'un vecteur(*Vector<?>*)

```
Vector <Personne> vecteurPersonnes ;
  /*
   * declaration de vecteurPersonnes comme etant un Vector
   * generique de Personne
   */
  vecteurPersonnes = new Vector<Personne>();
  /*
   * instanciation de vecteurPersonnes
   */
  vecteurPersonnes.add(new Personne("nom1","prénom1",new Date()));
  vecteurPersonnes.add(new Personne("nom2","prénom2",new Date()));
  /*
   * Ajout de deux Personne
   */

  for (Personne p: vecteurPersonnes)
   System.out.println(p);
  /*
   * Utilisation de la boucle for each pour parcourir le vecteur
   */

  for (Iterator<Personne> iterator = vecteurPersonnes.iterator(); iterator.hasNext();)
   System.out.println(iterator.next());
  /*
   * Utilisation d'un iterateur pour parcourir le vecteur
   */
```

6.3. Les maps

Une map est une collection qui associe une clé à une valeur. La clé est unique, contrairement à la valeur qui peut être associée à plusieurs clés. Vous retrouverez le même principe avec les tableaux associatifs en PHP. Map est une interface qui possède plusieurs classes qui l'implémente parmis lesquelles nous trouvons : *HashMap* qui acceptent les null et les *HashTable* qui ne les acceptent pas. Le code suivant permet de créer une HashTable de compte bancaire est d'accéder à un de ces éléments.

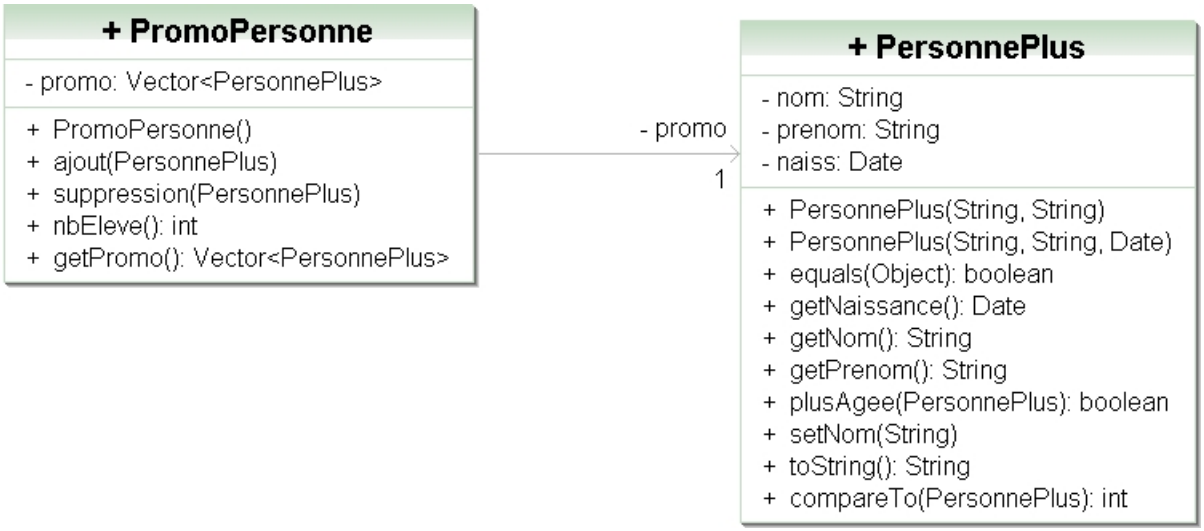
Exemple 4.9. Utilisation d'une *HashTable*

```
Hashtable <String,Personne> hashtablePersonnes;
  /*
   * Declaration de hastablePersonnes comme etant une HashTable
   * avec une String comme clef et
   * une Personne comme valeur
   */

  hashtablePersonnes = new Hashtable<String, Personne>();
  hashtablePersonnes.put("nom1",new Personne("nom1","prénom1",new Date()));
  hashtablePersonnes.put("nom2",new Personne("nom2","prénom1",new Date()));
  /*
   * ajout de deux Personne avec les clefs "nom1" et "nom2"
   */
  System.out.println(hashtablePersonnes.get("nom1"));
  System.out.println(hashtablePersonnes.get("nom2"));

  for (Personne p : hashtablePersonnes.values())
   System.out.println(p);
  /*
   * Affichage des valeurs
   */
```

6.4. Mise en oeuvre

Nous allons mettre en oeuvre les collections au travers de deux exemples: une promotion de personnes et un tableau associatif qui a un nom associe une personne.

Figure 4.23. Diagramme de classe de la Promotion



6.4.1. La promotion

Créer en respectant l'API la classe PromoPersonne. Au lieu d'utiliser Personne, vous utiliserez la classe PersonnePlus de tp.prof.classes. La classe PersonnePlus et la même que la classe Personne mais enrichie avec deux méthodes qui permettent la comparaison : `public boolean equals(Object obj)`,`public int compareTo(PersonnePlus o)`.

La première méthode permet la comparaison de deux personnes et la seconde le trie.

La première peut vous être utile pour supprimer un élément de la promotion :

```
public boolean equals(Object obj) {
  if (this == obj)
   return true;
```

```
 if (! (obj instanceof PersonnePlus))
  return false;
 PersonnePlus personne = (PersonnePlus) obj;
 return this.naiss.equals(personne.naiss) &&
 this.nom.equals(personne.nom) &&
 this.prenom.equals(personne.prenom);
}
```

Il est possible d'utiliser des méthodes de tri sur les collections. Pour cela les objets doivent être comparables. C'est pourquoi la classe *PersonnesPlus* est modifiée pour rendre ses instances *comparables*, ce qui suit n'est donné qu'à titre indicatif, nous reverrons les concepts en programmation :

1. PersoonePlus et modifiée pour qu'elle implémente l'interface *Comparable*

   ```
   public class PersonnePlus implements Comparable<PersonnePlus>{
   ```

2. Pour être `Comparable` il faut implémenter la méthode compareTo

   ```
   public int compareTo(PersonnePlus o) {
     int nomOk, prenomOk, naissOk;
     nomOk = this.nom.compareTo(o.nom);
     prenomOk = this.prenom.compareTo(o.prenom);
     naissOk = this.naiss.compareTo(o.naiss);
     if (nomOk == 0)
      if (prenomOk == 0)
       return naissOk;
      else
       return prenomOk;
     else
      return nomOk;
    }
   ```

Ceci étant fait vous pouvez utiliser la méthode de classe sort de Collections :

```
Collections.sort(....);
```

Cela devrait vous servir pour renvoyer un vecteur de PersonnesPlus triées.

6.4.2. Tableau associatif de personnes

Créer et tester un tableau associatif de Personnes dans la classe TestPersonnes qui a un nom (String) associe une Personne.

Chapitre 5. Devoir maison (le jeux de d'othello)

Vous allez réaliser un jeu d'othello, l'othello est un jeux où celui qui en fin de partie à le plus de pions à gagné. A son tour de jeu, le joueur doit poser un pion de sa couleur sur une case vide de l'othellier, adjacente à un pion adverse. Il doit également, en posant son pion, encadrer un ou plusieurs pions adverses entre le pion qu'il pose et un pion à sa couleur, déjà placé sur l'othellier. Il retourne alors de sa couleur le ou les pions qu'il vient d'encadrer. Les pions ne sont ni retirés de l'othellier, ni déplacés d'une case à l'autre. Un joueur qui ne peut jouer doit passer son tour. La partie est fini lorsque les joueurs ne peuvent plus jouer.

Figure 5.1. Taquin début du jeux

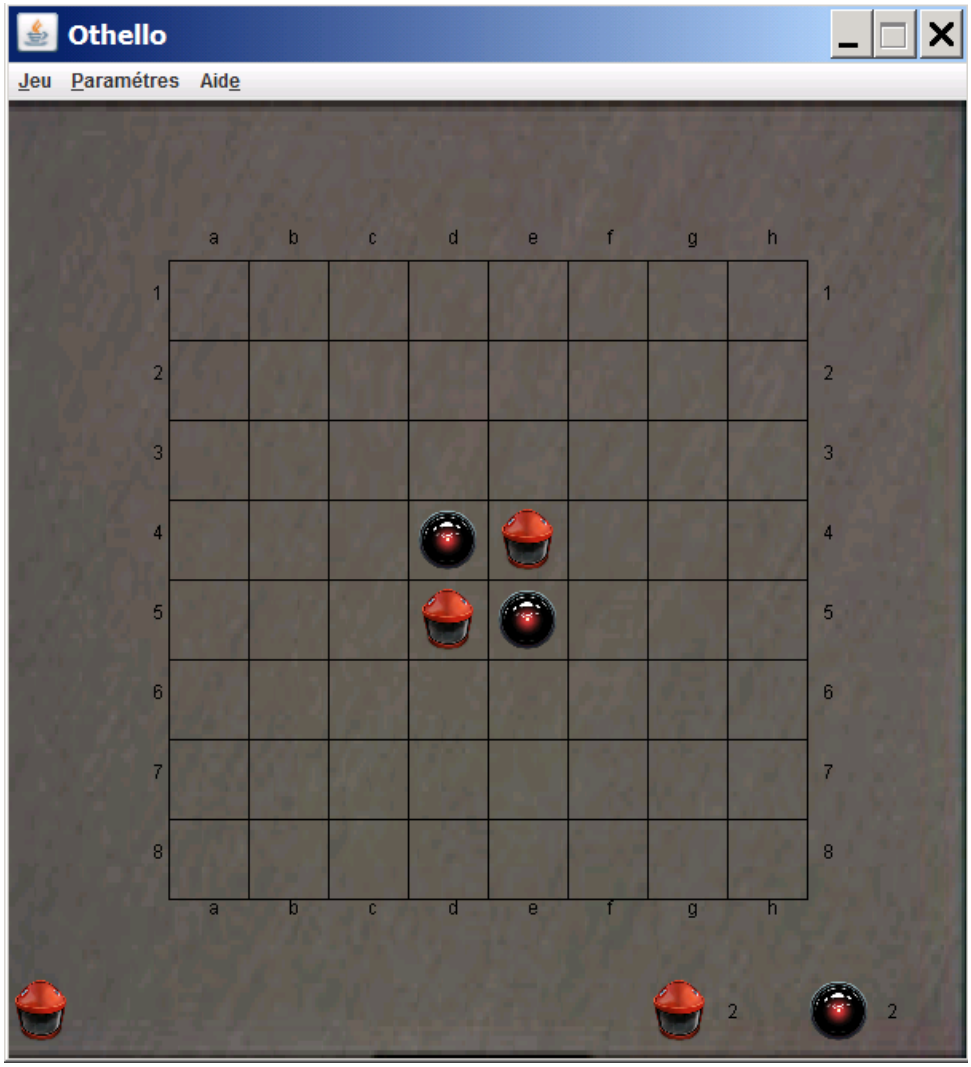
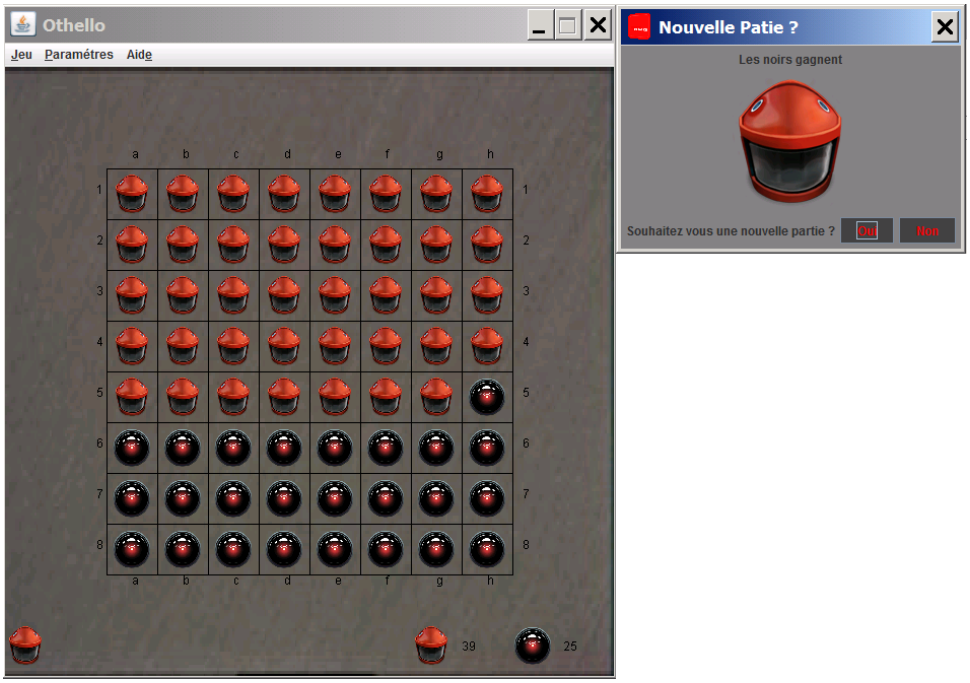


Figure 5.2. Othello partie gagnante



Pour avoir un rendu graphique nous allons décomposer notre jeux en trois catégories de classes :

- Les vues
- Les contrôleurs
- Les modèles

Vous n'aurez pas à gérer l'affichage (les vues) , ni à gérer la logique de contrôle (les contrôleurs) le code vous est fourni. Mais vous aurez à gérer un des modèle le *GameModel*. Un modèle est une représentation du monde, pour notre modèle nous utiliserons un tableau à deux dimensions qui représente l'othellier. Commençons donc avec une présentation des tableaux à deux dimensions.

1. Tableaux à deux dimensions

```
int[] t;
```

nous permet de définir *t* comme étant une référence vers un tableau d'entiers (à une dimension).

```
Object[] o;
```

nous permet de définir o comme étant un tableau d'objets. Les tableaux étant eux-même des objets, nous pouvons les stocker dans des tableaux d'objets et ainsi avoir des tableaux à deux dimensions. Le processus peut bien évidement être répété pour créer des tableaux à n dimensions.

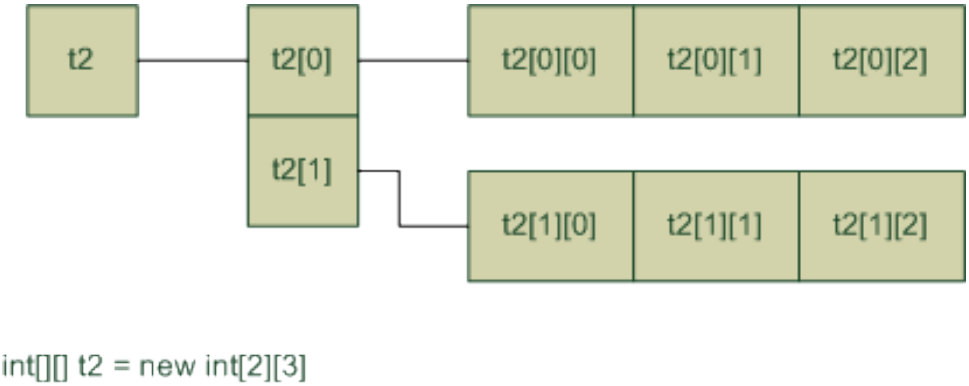
```
int[][] t2;
```

nous permet donc de déclarer un tableau d'entiers à 2 dimensions mais comment le créer.

```
int t2[][] = new int[2][3];
```

nous permet de définir et de créer un tableau de deux par trois.

Figure 5.3. Représentation physique d'un tableau de 2x3



int[][] t2 = new int[2][3]

Il existe d'autre possibilité qui permettent d'avoir des tableaux de différentes tailles comme :

```
int t2[][];
t2 = new int[2];
t2[0] = new int[3];
t2[1] = new int[2];
```

La représentation physique est indépendante de la représentation logique un même tableau physiquement stocké en mémoire peut avoir plusieurs représentations logiques est-il horizontal, vertical, l'élément [0][0] est-il en bas à gauche, en bas à droite, ...

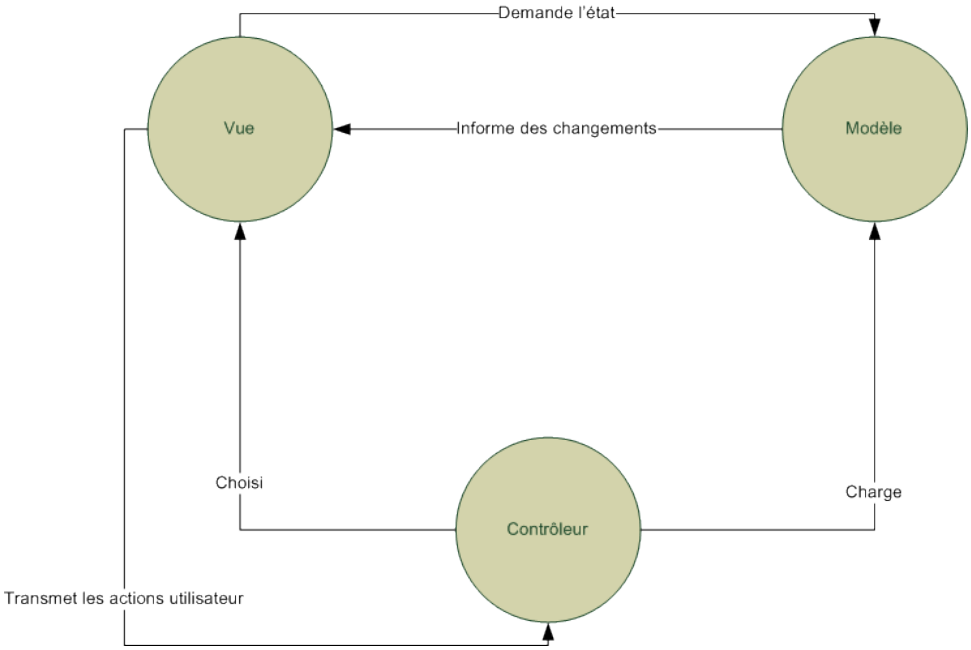
Figure 5.4. Représentation logique

2. Présentation du projet

Notre projet est composé de trois ensemble de classes : Vue, Contrôleur et Modèle. Vous n'aurez qu'a coder la classe *GameModele*, mais vous devez cependant respecter des règles de codage pour qu'elle puissent s'intégrer dans le reste de l'application. Vous devez donc coder les méthode *public* de *GameModel*. Les méthodes *private* sont données à titre indicatif, vous pouvez les coder ou non.

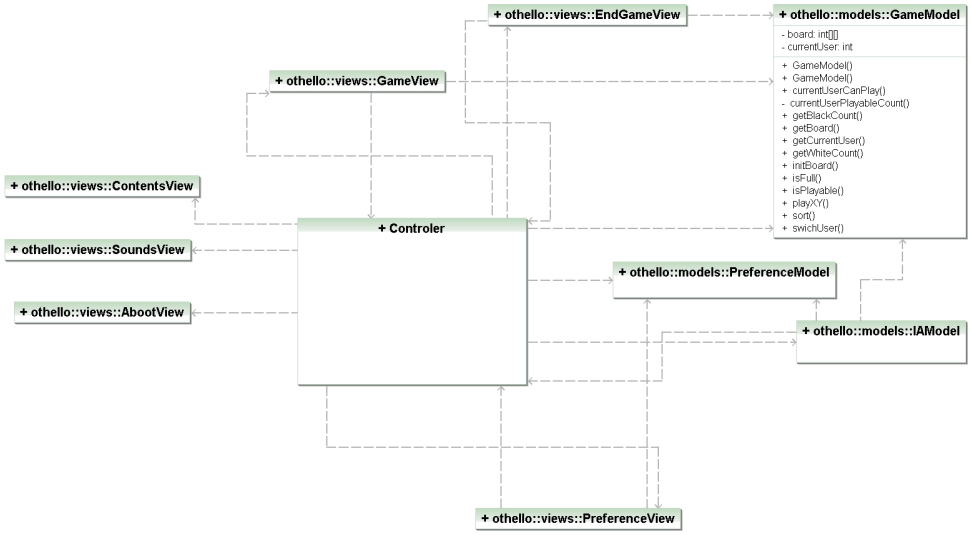
L'application repose sur un modèle qui sépare différents rôle, le patron de conception : Modèle, Vue, Contrôleur (MVC).

Figure 5.5. Modèle Vue Contrôleur



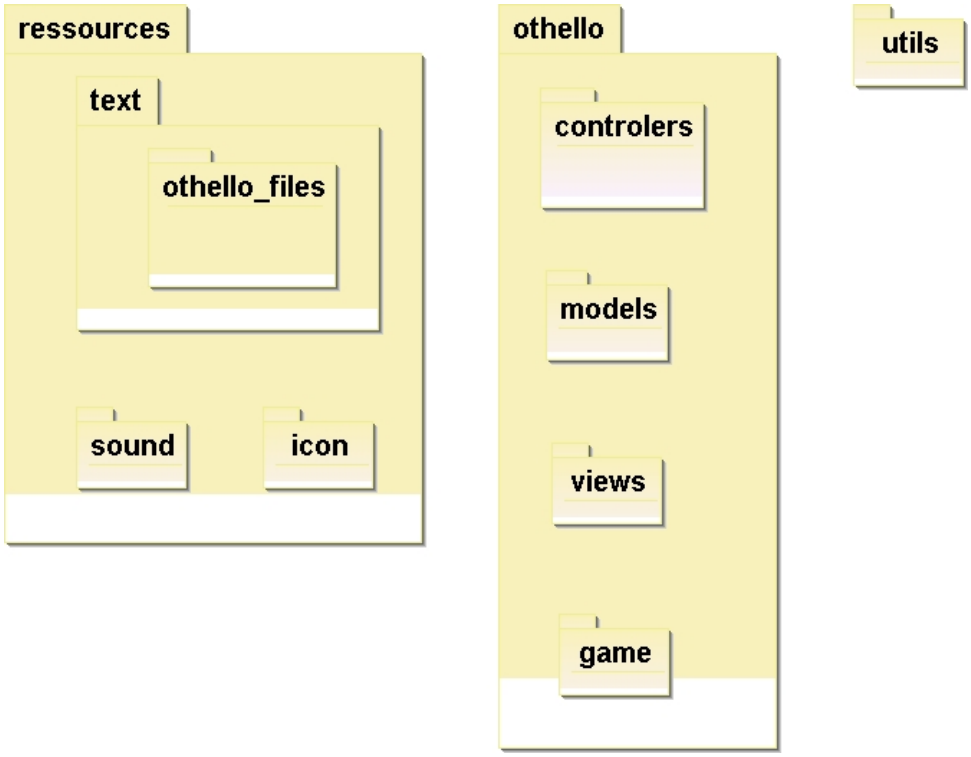
Ne disposant pas encore des outils ne programmation pour réaliser une bonne implémentation du MVC, nous allons en offrir une approche amoindrie. Le diagramme de classe de notre application est le suivant :

Figure 5.6. Diagramme de classe



La seule classe à compléter est celle qui est développée. La structure des paquetages est la suivante :

Figure 5.7. Paquetage de l'application



3. Travail a réaliser

Vous allez importer dans votre *worspace* le projet le contenu de l'archive *othello.jar*. Vous devez compléter la classe GameModele pour que le jeu soit fonctionnel. L'attribut *debug* de *GameBoard* peut-être mis à *true* pour afficher les numéros de casses.

La classe *Othello* contient le *main*.

Le travail est individuel est facultatif vous le travail doit être exporter puis rendu sur le serveur ftp ftp-exam.src avant le 22/04/10.

Annexe A. Import et export de projet sous eclipse

L'unité de travail sous eclipse est le projet, voici deux moyens pour échanger vos données en exportant et en important vos données.

1. Export

Pour exporter un projet, vous pouvez suivre la procédure suivante :

1. Clic-droit sur le projet puis "export"
2. choisir "general" et "archive file"
3. choisir les sources à exporter

2. Import d'un projet dans un Workspace existant

Pour importer un projet dans un workspace existant :

1. choisir "file" puis "import"
2. choisir "general" puis "existing projects into workspace"