M2.23.3 Algorithmique

Jean-François Remm
Jean-François Berdjugin
Vanda Luengo

M2.23.3 Algorithmique

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

Publié le 28 février 2009

Table des matières

Introduction vii
1. Travaux Dirigés 1
    1. Tableaux 1
        1.1. Définition 1
            1.1.1. Déclaration 1
            1.1.2. Instanciation (ou construction) 1
            1.1.3. Accès aux données 2
        1.2. Exemples 3
        1.3. Exercices 4
            1.3.1. Première exécution 4
            1.3.2. Deuxième exécution 4
            1.3.3. Troisième exécution 4
            1.3.4. Somme des éléments d'un tableau 5
            1.3.5. Position du maximum 5
            1.3.6. Maximum 5
            1.3.7. Inversion 5
            1.3.8. Insertion 5
            1.3.9. Suppression 5
            1.3.10. Fusion 5
            1.3.11. Recherche séquentielle dans un tableau non trié 5
            1.3.12. Recherche séquentielle dans un tableau trié 5
            1.3.13. Recherche dichotomique dans un tableau trié (facultatif) 5
    2. Récursivité 6
        2.1. Définition 6
        2.2. Exemples 6
            2.2.1. Factorielle 6
            2.2.2. Somme d'un tableau d'entiers 7
        2.3. Exercices 8
            2.3.1. Fibonacci 8
            2.3.2. Position du maximum 9
    3. Classe et Objet 9
        3.1. Définitions 9
        3.2. Exemple 9
            3.2.1. Point 9
        3.3. Exercices 15
            3.3.1. Classe Cercle 15
            3.3.2. Classe Vehicule 16
            3.3.3. Classe Personne et Classe Compte Bancaire 16
    4. Cellule 17
        4.1. Présentation 17
        4.2. Codage 17
    5. Liste simplement chaînée 18
        5.1. Présentation 18
        5.2. Codage 18
            5.2.1. Variable(s) d'instance(s) 18
            5.2.2. Constructeur sans paramètre 19
            5.2.3. Méthode estVide 19
            5.2.4. Méthode getPremier 19
            5.2.5. Méthode getDernier 19
            5.2.6. Méthode insereTete 19
            5.2.7. Méthode insereQueue 19
            5.2.8. Méthode suppressionTete 19
            5.2.9. Méthode suppressionQueue 19
            5.2.10. Exercices complémentaires 19
    6. Pile 20

6.1. Présentation ... 20
6.2. Codage ... 20
2. Travaux pratiques ... 21
1. Présentation de l'IDE Eclipse et mise en place de l'espace de travail ... 21
1.1. Mise en place de l'espace de travail ... 21
1.2. Premier test ... 22
2. Tableaux ... 22
2.1. Initialisation des tableaux ... 23
2.1.1. Affichage des éléments d'un tableau ... 23
2.1.2. Lecture interactive ... 23
2.1.3. Initialisation aléatoire ... 23
2.2. Somme des éléments d'un tableau ... 24
2.3. Position du maximum ... 24
2.4. Recherche séquentielle dans un tableau non trié ... 24
2.5. Recopie d'un tableau ... 24
2.6. Tri ... 24
2.7. Recherche dichotomique dans un tableau trié ... 24
2.8. Égalité entre deux tableau ... 24
3. Classes et objets ... 24
3.1. Utilisation d'une API ... 24
3.1.1. Application Programming Interface ... 25
3.1.2. Vehicule ... 26
3.1.3. Point ... 27
3.1.4. String et StringBuffer ... 27
3.1.5. Date ... 28
3.2. Création d'une classe ... 29
3.2.1. Point ... 29
3.2.2. Personne ... 29
3.3. Variables de classe ... 30
3.3.1. Rappels ... 30
3.3.2. Classe CompteBancaire ... 31
4. Collections ... 32
4.1. Présentation ... 32
4.2. Listes ... 33
4.3. Les maps ... 34
4.4. Mise en oeuvre ... 35
4.4.1. La promotion ... 35
4.4.2. Tableau associatif de personnes ... 36
5. Récursivité ... 36
3. Devoir maison (le jeux de d'othello) ... 37
1. Tableaux à deux dimensions ... 38
2. Présentation du projet ... 40
3. Travail a réaliser ... 42
A. Import et export de projet sous eclipse ... 43
1. Export ... 43
2. Import d'un projet dans un Workspace existant ... 43
Glossaire ... 44

Liste des illustrations

1.1. Déclaration d'un tableau ... 1
1.2. Construction d'un tableau ... 1
1.3. Accès aux éléments d'un tableau ... 2
1.4. Dépassement des bornes ... 3
1.5. Alias ... 3
1.6. Appel récursifs de la factorielle ... 7
1.7. Appel récursif de la somme des éléments d'un tableau d'entiers ... 8
1.8. Appel récursif de la somme des éléments d'un tableau d'entiers avec un index. ... 8
1.9. Diagramme de classe de Point ... 10
1.10. Teste d'un point ... 11
1.11. Diagramme des séquences ... 12
1.12. Association entre Point et Cercle ... 16
1.13. Classe Personne ... 17
1.14. Cellule ... 17
1.15. Diagramme de classe de Cellule ... 17
1.16. Liste simplement chaînée ... 18
1.17. Liste simplement chaînée réelle ... 18
1.18. Diagramme UML de la classe Liste ... 18
1.19. Diagramme de classe de Pile ... 20
2.1. Workspace ... 22
2.2. Classe véhicule ... 26
2.3. Classe Point ... 27
2.4. Classe Point ... 29
2.5. Classe Personne ... 29
2.6. Comptebancaire ... 32
2.7. Collections ... 32
2.8. Liste ... 33
2.9. Diagramme de classe de la Promotion ... 35
3.1. Taquin début du jeux ... 37
3.2. Othello partie gagnante ... 38
3.3. Représentation physique d'un tableau de 2x3 ... 39
3.4. Représentation logique ... 40
3.5. Modèle Vue Contrôleur ... 41
3.6. Diagramme de classe ... 41
3.7. Paquetage de l'application ... 42

Liste des exemples

1.1. Affichage de voyelles 3
1.2. Factorielle récursive 6
1.3. Somme récursive des éléments d'un tableau d'entiers 7
1.4. Utilisation de la classe Point 13
2.1. Tableau des carrés initialisé avec une boucle 23
2.2. Tableau des carrés initialisé avec une liste de valeur 23
2.3. Variable de classe 30
2.4. Variable de classe constante 30
2.5. Initialisation et déclaration d'une variable de classe constante 31
2.6. Bloc d'initialisation static 31
2.7. Utilisation d'un vecteur 34
2.8. Utilisation d'une HashTable 34

Introduction

Nous disposons, ce semestre, de 6 séances d'une heure et demie de Travaux Dirigés (TD) et de 6 séances de Travaux Pratiques (TP) de 3 heures. Avec ce volume horaire, nous devons aborder les structures de données qui sont un ensemble de données accessible via un ou des noms. La première des structures que nous aborderons sera le tableau.

Ayant fini le semestre précédant par le langage Java, nous continuerons ce semestre à l'utiliser pour l'enseignement de l'algorithmique que ce soit sur papier ou sur machine.

Chapitre 1. Travaux Dirigés

1. Tableaux

Les tableaux sont la première structure de donnée que nous allons aborder. Cette structure est ancienne et d'un usage courant dans de nombreux langages de programmation. Pour les tableaux simples, les éléments de la structure sont simplement accédés via un indice. *Vous utiliserez l'année prochaine des tableaux (associatifs) en PHP (Hypertext Preprocessor) pour récupérer les valeurs saisies dans les formulaires.*

En java les tableaux sont des objets particuliers, mais dans ce TD d'algorithmique nous pouvons les utiliser sans notions d'objets, ce que nous allons faire.

1.1. Définition

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.

Nous allons apprendre comment déclarer un tableau, puis l'instancier et enfin le manipuler.

1.1.1. 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 1.1. Déclaration d'un tableau

tab

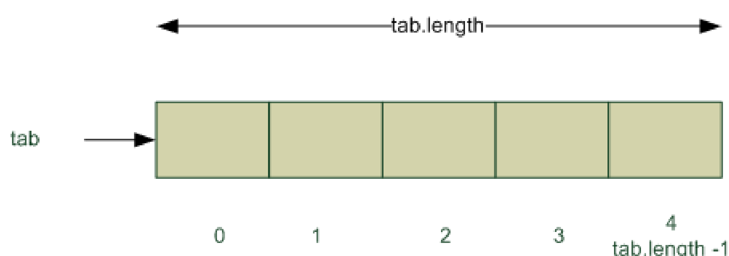
1.1.2. Instanciation (ou construction)

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 1.2. Construction d'un tableau



1.1.3. Accès aux données

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

1.1.3.1. Lecture/Ecriture

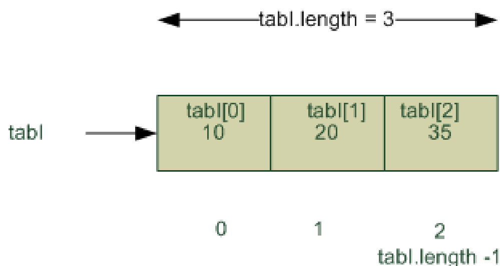
La lecture et l'écriture sont réalisées en spécifiant entre crochet (*[]*) la position de la "case".

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 1.3. Accès aux éléments d'un tableau



1.1.3.2. 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
```

1.1.3.3. Pièges

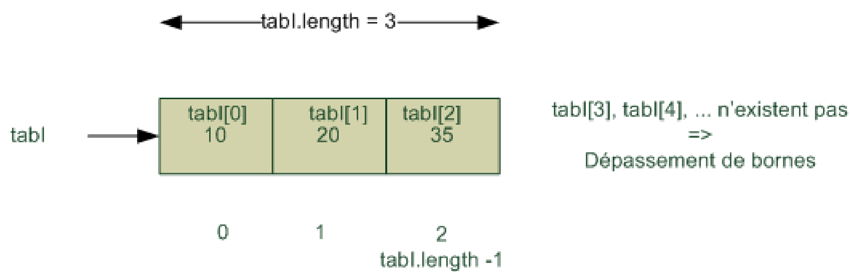
Les tableaux contiennent deux pièges intrinsèques : le dépassement des bornes et les alias involontaires.

1.1.3.3.1. 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.

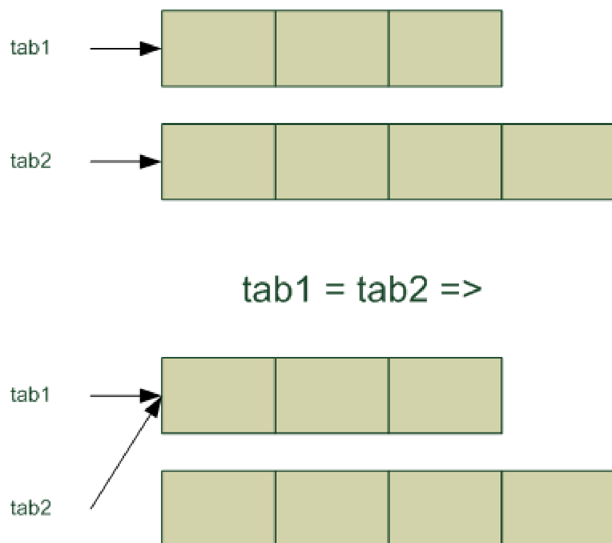
Figure 1.4. Dépassement des bornes



1.1.3.3.2. 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 1.5. Alias



1.2. Exemples

Exemple 1.1. Affichage de voyelles

```
public class Exemple1{
  public static void main(String[] args){
    char[] voyelle; //declaration de voyelle comme étant une référence sur un tableau de caractères
    voyelle=new char[6]; //construction d'un tableau de 6 caractères référencé par voyelle
    voyelle[0]='a';
    voyelle[1]='e';
    voyelle[2]='i';
    voyelle[3]='o';
    voyelle[4]='u';
    voyelle[5]='y';
    for (int i=0; i < voyelle.length; i++){
       System.out.println(voyelle[i]);
    }
  }
}
```

Cet exemple permet de créer et d'afficher un tableau de caractères contenant des voyelles.

1.3. Exercices

Non allons travailler en deux phases : la première est le résultat d'exécution et la seconde et l'écriture de code. Pour l'écriture du code, nous utiliserons des méthodes de classe (*static*).

1.3.1. Première exécution

Exécuter l'algorithme suivant :

```
public class Exercice1{
  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]);
    }
  }
}
```

1.3.2. Deuxième exécution

Exécuter l'algorithme suivant :

```
public class Exercice2{
  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]);
    }
  }
}
```

1.3.3. Troisième exécution

Exécuter l'algorithme suivant :

```
public class Exercice3{
  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]);
    }
  }
}
```

[1] La généricité nous permettra plus tard de réaliser des traitements indépendamment du type.

1.3.4. Somme des éléments d'un tableau

Pour cet exercice et pour les suivants nous allons travailler sur des tableaux d'entiers. Nous supposons l'existence d'une classe (utilitaire) TableauInt dans laquelle nous allons placer des méthodes de classe pour la manipulation de tableaux d'entiers.[1]

La première méthode n'est pas à écrire mais a compléter, c'est la méthode somme qui calcule la somme des éléments d'un tableau. Le début du code est le suivant :

```
public class TableauInt{
  public static int somme(int[] t){
    //Le code se place ici
  }
}
```

Compléter la méthode à l'endroit indiqué.

1.3.5. Position du maximum

Écrire une méthode de classe qui prend un tableau d'entiers en paramètre et détermine la position du plus grand élément de celui-ci.

1.3.6. Maximum

Écrire une méthode de classe qui prend un tableau d'entiers en paramètre et détermine le plus grand élément de celui-ci

1.3.7. Inversion

Écrire une méthode de classe qui permet d'inverser un tableau d'entiers. Il ne faut pas créer un nouveau tableau mais inverser le tableau reçu en paramètre.

1.3.8. Insertion

Écrire une méthode de classe qui *insère un entier* passé en paramètre dans un tableau à une *position* donnée. *Le dernier élément sera perdu.*

1.3.9. Suppression

Écrire une méthode de classe qui supprime l'élément à la position pos dans un tableau d'entier. *Peu importe la dernière valeur du tableau.*

1.3.10. Fusion

Écrire une méthode de classe qui prend en paramètre deux tableaux et renvoie le tableau résultant de leur concaténation. Le tableau résultant doit-être créé dans la méthode.

1.3.11. Recherche séquentielle dans un tableau non trié

Écrire une méthode de classe qui prend en paramètre un tableau d'entiers et une valeur et qui renvoie la position (première trouvée) de la valeur ou -1 si cette valeur n'a pas été trouvée dans le tableau.

1.3.12. Recherche séquentielle dans un tableau trié

On peut arrêter la recherche quand la valeur recherché ne peut plus être trouvée. On suppose le tableau trié de manière croissante. Le type des éléments du tableau doit être doté d'un ordre total (i.e. # elt1, elt2 : type elt1# elt2 ou elt1 < elt2).

1.3.13. Recherche dichotomique dans un tableau trié (facultatif)

Le principe de cette recherche est très astucieux (et économique). Algo en log(n).*Les algorithme de tri seront vus en cours et en TP.*

2. 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.

2.1. 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.

2.2. 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.

2.2.1. Factorielle

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

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

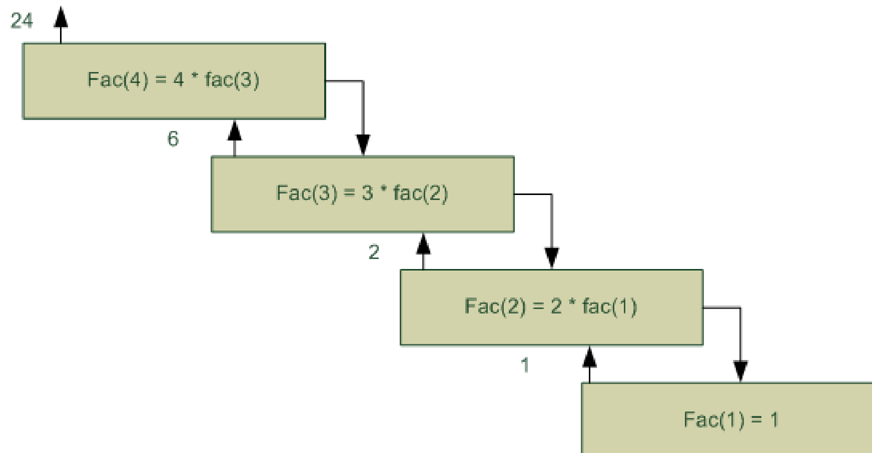
La première version permet l'implantation d'une fonction itérative. La deuxième permet une implantation récursive de la fonction factorielle.

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

Exemple 1.2. 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 1.6. Appel récursifs de la factorielle



2.2.2. Somme d'un tableau d'entiers

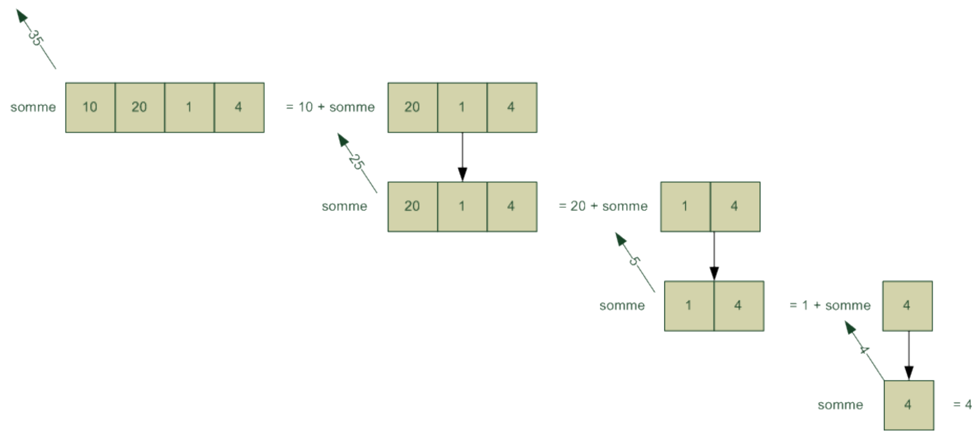
Le cas d'arrêt est le tableau à une case, nous pouvons conclure que sa somme est celle de sa case. L'appel récursifs est obtenu en constatant que la somme des éléments est égale au premier élément plus la somme du sous tableau restant. Nous supposons la méthode extraction qui permet d'extraire un sous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

```
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 1.3. 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;
 }
```

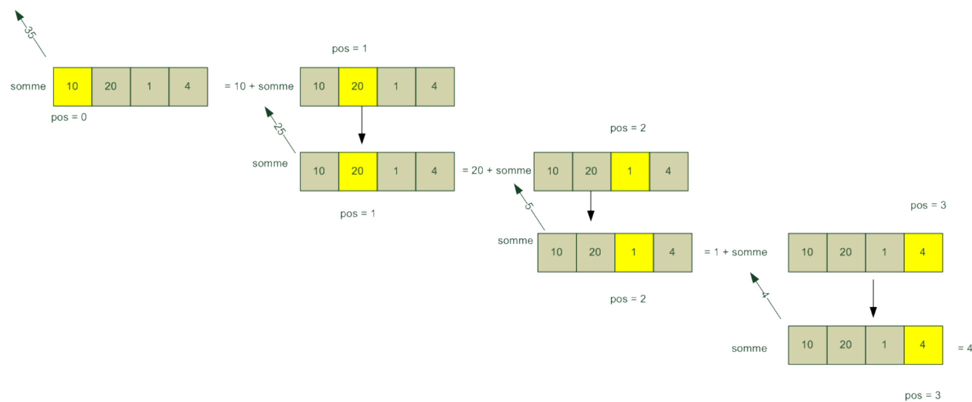
Figure 1.7. Appel récursif de la somme des éléments d'un tableau d'entiers



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



2.3. Exercices

Nous allons dans un premier temps réaliser un appel récursif sur la suite de Fibonacci puis donner une version récursive de la position du maximum.

2.3.1. Fibonacci

La suite de Fibonacci est définie comme suit :

- $U_0 = 1$
- $U_1 = 1$

- $U_n = U_{n-1} + U_{n-2}$

ce qui nous donne :

```
public static int fibonacci(int n)
 {
  int res = 1;
  if (n > 1)
  {
   res = fibonacci(n-1) + fibonacci(n-2);
  }
  return res;
 }
```

Donner l'appel récursif de Fibonacci de 4.

2.3.2. Position du maximum

Écrire en utilisant la récurrence une méthode position du maximum. Dans un soucis de performance, on ne va pas créer un tableau extrait mais déplacer un curseur pour délimiter un sous tableau :

```
public static int posMaxR(int[] t, int pos)
```

Je vous conseil d'exprimer votre idée oralement avant tout codage, les pièges sont nombreux.

3. Classe et Objet

Les tableaux que nous avons vu sont des objets accessibles via une syntaxe particulière.

3.1. Définitions

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 .

3.2. Exemple

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

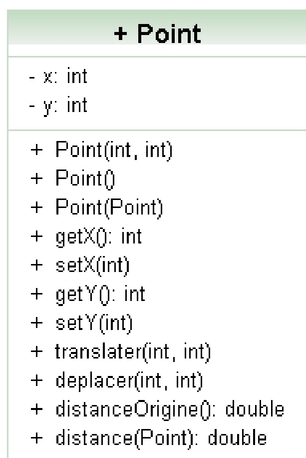
3.2.1. Point

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.

3.2.1.1. Représentation UML

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 1.9. 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) : *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.
  - 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
      - distanceOrigine():double et distance(Point):double qui donne la distance à l'origine et la distance à 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 des objets points et les utiliser.

3.2.1.2. Utilisation

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 1.10. Teste d'un point

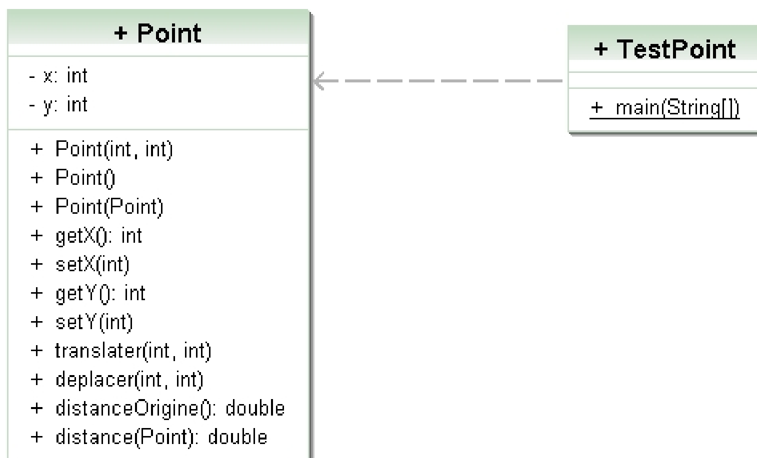
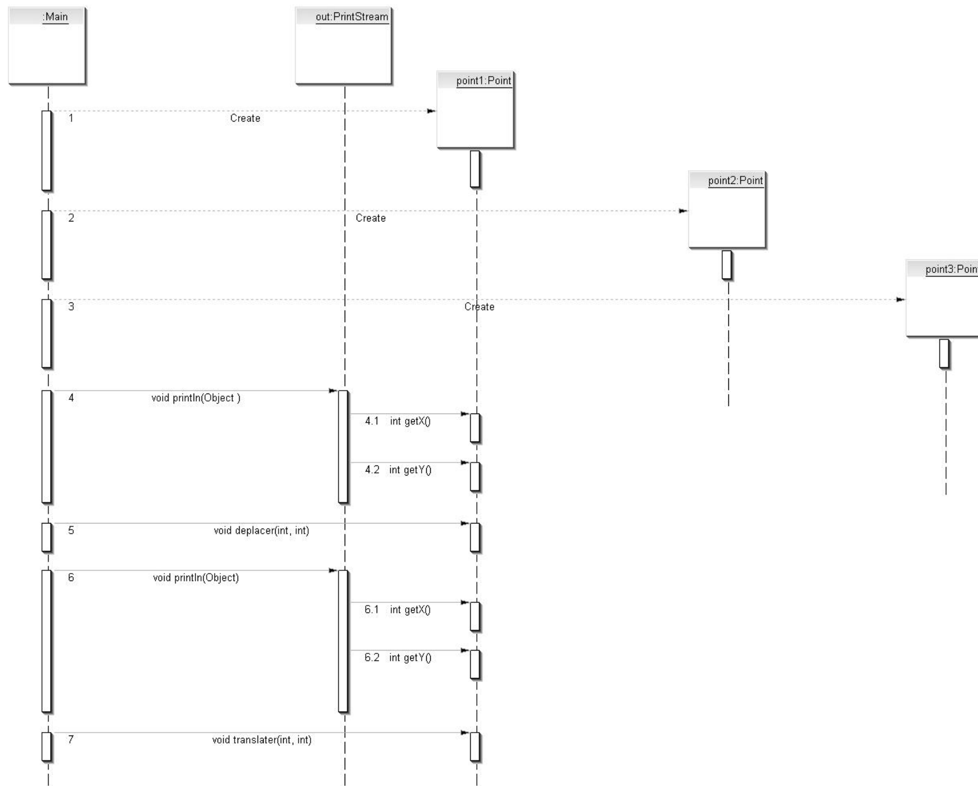


Figure 1.11. Diagramme des séquences



Exemple 1.4. 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 de 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
  System.out.println(point1.distanceOrigine());
                             //12.0
  System.out.println(point1.distance(point2));
                             //11.180339887498949
  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.

3.2.1.3. Codage

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 td;

/**
 * @author jub
 *Classe d'un point 2D
 */

public class Point {

/**
 * x: abscisse
```

```
 * y: ordonnée
 */

private int x,y;

/**
 * Permet de créer un point de coordonnées (x,y)
 * @param x l'abscisse
 * @param y l'ordonnée
 */
public Point(int x, int y) {
 this.x = x;  //this.x permet de manipuler la variable d'instance x
 this.y = y;  //this.y permet de manipuler la variable d'instance y
}

/**
 * Permet de créer un point de coordonnées (0,0)
 */
public Point() {
 this(0,0);  //this(0,0) permet d'appeler le constructeur qui prend en parametre deux entiers
}

/**
 * Permet de créer un point de mêmes coordonnées qu'un autre point
 * @param point
 */
public Point(Point point) {
 this(point.getX(),point.getY());
}

/**
 * Renvoie l'abscisse
 * @return the x
 */
public int getX() {
 return x;
}

/**
 * Définit l'abscisse
 * @param x the x to set
 */
public void setX(int x) {
 this.x = x;
}

/**
 * Renvoie l'ordonnée
 * @return the y
 */
public int getY() {
 return y;
}

/**
 * Définie l'ordonnée
 * @param y the y to set
 */
public void setY(int y) {
 this.y = y;
}



/**
 * translate le point courant de dx et dy (i.e les nouvelles coordonnées de mon Point sont (x+dx,y+dy) )
 * @param dx un int
 * @param dy un int
 */
public void translater(int dx,
```

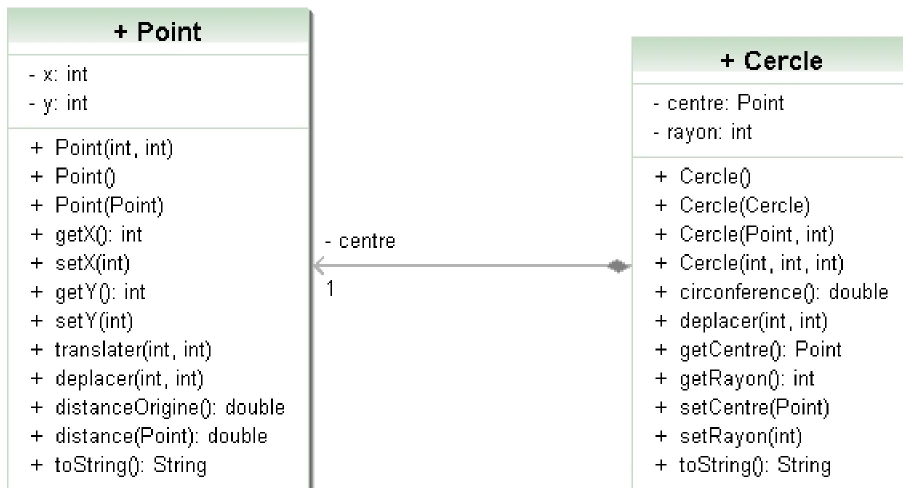
```
                       int dy)
{
 this.setX(this.getX()+dx);
 this.setY(this.getY()+dy);
}


/**
 * déplace le point courant en (newX,newY)(i.e les nouvelles coordonnées de mon Point sont (newX,newY) )
 * @param x un int
 * @param y un int
 */
public void deplacer(int x,
                     int y)

{
 this.setX(x);
 this.setY(y);
}


/**
 * renvoie la distance à l'origine du Point courant
 * @return distance à l'origine
 */
public double distanceOrigine()
{
 return this.distance(new Point(0,0));
}

/**
 * Renvoie la distance entre deux points
 * @param autrePt
 * @return distance
 */
public double distance(Point autrePt)
{
 double dx,dy;
 dx=this.getX()-autrePt.getX();
 dy=this.getY()-autrePt.getY();

 double res;
 res= Math.sqrt(Math.pow(dx, 2)+ Math.pow(dy,2));
 return res;
}


/* (non-Javadoc)
 * @see java.lang.Object#toString()
 */
public String toString()
{
 String s="("+this.getX()+","+this.getY()+")";
 return s;
}


}
```

3.3. Exercices

3.3.1. Classe Cercle

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

Figure 1.12. 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 créer la classe Cercle vous allez l'utiliser pour :

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. Ne faudrait-il pas une nouvelle méthode et quelle serait sa signature[2] ?

Proposer maintenant une implémentation de la classe Cercle.

3.3.2. Classe Vehicule

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[3].

3.3.3. Classe Personne et Classe Compte Bancaire

Un compte bancaire qui est détenu par un titulaire peut se débiter, se créditer. On peut consulter le solde d'un compte bancaire. La classe Personne est fournie, donner le diagramme de classe puis l'implémentation de compte bancaire.

[3]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 ...

Figure 1.13. Classe Personne

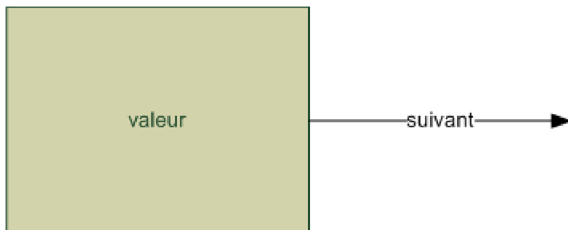
4. Cellule

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

4.1. Présentation

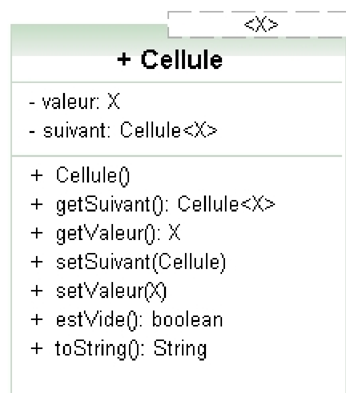
Une cellule est définie par une valeur et une référence vers une autre cellule.

Figure 1.14. Cellule



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

Figure 1.15. Diagramme de classe de Cellule



Le que vous observé est un type passé en paramètre, ainsi la classe Cellule est indépendante du type, on parle de généricité.

4.2. Codage

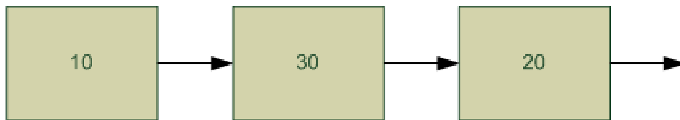
Donner le code la classe Cellule sachant que le constructeur construit une cellule vide.

5. Liste simplement chaînée

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.

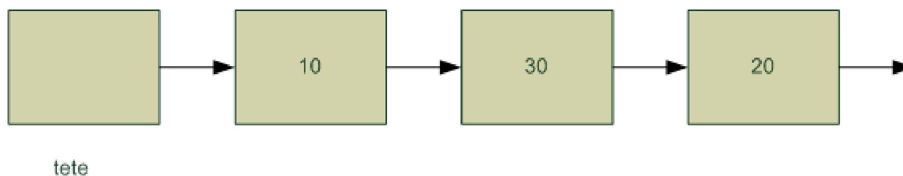
Figure 1.16. Liste simplement chaînée



5.1. 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.

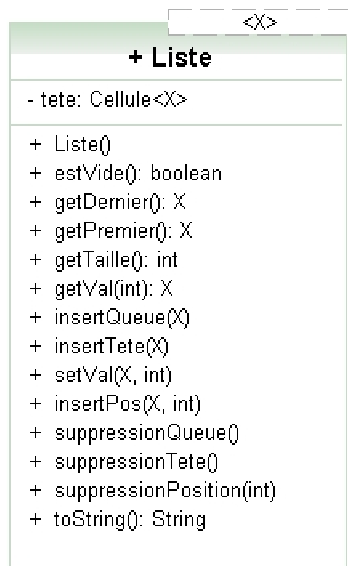
Figure 1.17. Liste simplement chaînée réelle



5.2. Codage

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

Figure 1.18. Diagramme UML de la classe Liste



Donner pour chacune des étapes qui vont suivre le code java correspondant.

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

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

5.2.2. Constructeur sans paramètre

Donner le code java correspondant.

5.2.3. Méthode estVide

Une méthode pour savoir si la liste est vide.

5.2.4. Méthode getPremier

Accès au premier élément de la liste.

5.2.5. Méthode getDernier

Accès au dernier élément de la liste.

5.2.6. Méthode insereTete

On insère une nouvelle valeur en tête de liste.

5.2.7. Méthode insereQueue

On insère une nouvelle valeur en queue de liste.

5.2.8. Méthode suppressionTete

On supprime la valeur en tête de liste.

5.2.9. Méthode suppressionQueue

On supprime la valeur en queue de liste

5.2.10. Exercices complémentaires

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

5.2.10.1. 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.

5.2.10.2. Méthode getVal

Renvoie l'élément à une position quelconque.

5.2.10.3. Méthode setVal

Modifie un élément à une position quelconque.

5.2.10.4. Méthode insertPosition

Insère un élément à une position quelconque.

5.2.10.5. Méthode suppressionPosition

Supprime l'élément à une position quelconque.

6. 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.

6.1. 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 1.19. Diagramme de classe de Pile

6.2. Codage

Donner le code de la classe Pile.

Chapitre 2. Travaux pratiques

Pour réaliser ces TP, vous disposez de 6 séances de trois heures. Nous utiliserons l'environnement de développement eclipse, si vous souhaitez l'installer, chez vous, vous pouvez le télécharger à l'URL suivante : `http://www.eclipse.org/` . Nous allons aborder la notion de structure de données ainsi que les algorithmes associés (tri, parcours, mise à jour) puis enfin la récursivité.

En programmation un type de données, ou simplement type, définit le genre de contenu d'une donnée et les opérations pouvant être effectuées sur la variable correspondante. Les structures de données sont des structures logiques destinées à contenir des données, afin de leur donner une organisation permettant de simplifier leur traitement. Nous avons verrons les tableaux puis les classes et enfin une introduction aux collections java.

1. Présentation de l'IDE Eclipse et mise en place de l'espace de travail

Eclipse est un Integrated Development Environment (IDE), un environnement de développement comprenant un éditeur de texte, un compilateur, des outils automatiques de fabrication et un débogueur. Un autre IDE disponible est NetBeans de Sun (`http://netbeans.org/`).

Eclipse permet de gérer plusieurs *projets* qui seront stockés dans un *workspace*. Les projets seront pour nous des projets java qui contiendrons des *paquetages*. Les paquetages java permettent de structurer les applications, ils contiennent les *classes* (et les *interfaces*), ils constituent un espace de nom, ainsi deux classes peuvent avoir le même nom, si elles ne sont pas le même paquetage.

1.1. Mise en place de l'espace de travail

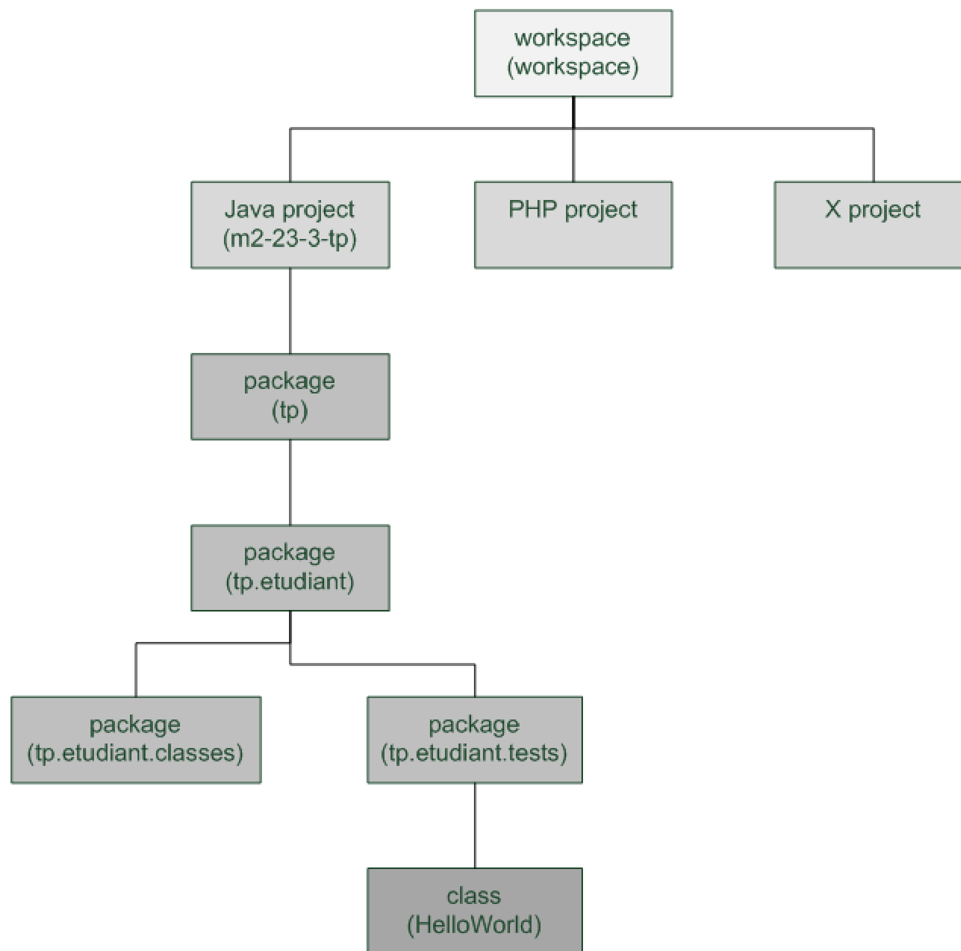
Vous allez lancer eclipse et suivre la démarche suivante :

1. Créer un dossier "*workspace*_m2.23.3" sur le partage netBios du serveur (`z:`).
2. Se rendre sur le plan de travail (workbentch).
3. Importer dans le workspace le projet existant : `m2-23-3-tp` le projet est contenu dans l'archive : `m2-23-3-tp.zip`.
4. Dans le projet m2-23-3 vous trouverez les paquetages suivants : `tp.etudiant.classes` et `tp.etudiant.tests`, ce sont les paquetages avec lesquels vous travaillerez. Pour le moment, ils contiennent les classes pour la récurrence, vous allez bientôt les faire grandir.
5. Le projet contient aussi une librairie externe (`lib`) contentant le paquetage `tp.prof.classes`, ce paquetage contiennent uniquement les `.class` des classes que vous aurez à tester (Point et Vehicule).
6. Pour finir le projet contient la javadoc des classes du paquetage `tp.prof.classes`. Une javadoc est la documentation d'une API (Application Programming Interface) en HTML obtenue grâce à des commentaires du code. Vous aurez à suivre cette documentation pour utiliser la classePoint, la classe Vehicule et créer les classes de l'énoncé.

Astuce

Vous pouvez accéder à la javadoc en positionnant le curseur sur l'élément choisi et en utilisant **SHIFT+F2** sur ce même élément.

Figure 2.1. Workspace



1.2. Premier test

Nous allons réaliser notre première application HelloWorld qui affiche ce message à l'écran :

1. Dans le paquetage *tp.etudiant.tests*, créer une nouvelle classe nommée *HelloWorld* en sélectionnant la création d'une méthode *main*.

2. A la place de

   ```
   // TODO Auto-generated method stub
   ```

   mettre

   ```
   System.out.println("Hello World");
   ```

   Vous constaterez que l'IDE vous guide avec des menus déroulant. De même vous pouvez obtenir des suggestions de nommage ou des complétions avec **CTRL**+**ESPACE**.

3. Pour exécuter votre application, vous allez dans les propriétés de la classe choisir run as (**Alt**+**Shift**+**Xj**). Dans la fenêtre de console, vous devriez observer le résultat.

Nous pouvons commencer notre programmation en apprenant à utiliser des tableaux.

2. Tableaux

Ce TP a pour but de vous faire utiliser des tableaux. Pour cela, nous reprenons des problèmes déjà abordés dans les TD. Vous devez créer dans le paquetage *tp.etudiant.classes* une classe *utilitaire TableauInt* qui contiendra tout

les méthodes statiques nous permettant de manipuler nos tableaux. Dans le paquetage *tp.etudiant.tests* vous aller créer une classe *TestTableau* qui contiendra la méthode *main*.

2.1. Initialisation des tableaux

Un tableau est un "objet" particulier dont les "getters and setters" sont les [ ]. Comme tous les objets, il doit être déclaré puis instancié.

```
int[] t; //t est déclaré comme étant un tableau d'entiers.
t = new int[6]; // t est instancié comme étant un tableau de 6 cases de 0 à 5
```

Pour manipuler les éléments d'un tableau nous devons indiquer la position :

```
t[4]=10; //la valeur 10 est rangée à la position 4.
int n=t[4]; //n reçoit la valeur de l'élément de position 4.
```

Pour initialiser notre tableau, nous avons plusieurs possibilités :

faire une boucle

L'exemple suivant construit un tableau est lui affecte le carré de la position. L'attribut *length* vous permet de connaître la taille de votre tableau.

Exemple 2.1. Tableau des carrés initialisé avec une boucle

```
int[] t = new int[4];
for (int i=0; i < t.length; i++){
  t[i]=i*i;
}
```

initialiser le tableau à sa création

Les *{ }* permettent d'initialiser un tableau à sa création en spécifiant une liste de valeur, l'exemple suivant à un comportement identique à l'exemple précédent :

Exemple 2.2. Tableau des carrés initialisé avec une liste de valeur

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

2.1.1. Affichage des éléments d'un tableau

Écrire "affiche" une méthode de classe qui prend en paramètre un tableau et qui affiche ses éléments :

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

puis tester la méthode.

2.1.2. Lecture interactive

```
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.

2.1.3. Initialisation aléatoire

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;
```

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

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

2.2. 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)
```

2.3. Position du maximum

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

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

2.4. 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)
```

2.5. 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)
```

2.6. 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)
```

2.7. 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)
```

2.8. Égalité entre deux tableau

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

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

3. Classes et objets

3.1. Utilisation d'une API

Ce TP a pour but de vous faire créer des classes de tests, en manipulant des objets définis par des classes que nous avons élaborées et dont nous vous fournissons les API (*Vehicule*, *Point*) ou qui figurent dans l'API standard Java (*String* et *StringBuffer* et *Date*). Attention les fichiers `Vehicule.class` et *Point.class* devront être importés dans votre projet (ce qui n'est bien entendu pas le cas des fichiers `String.class`, *StringBuffer.class* et *Date.class* qui sont disponibles par défaut depuis n'importe quel projet).

3.1.1. Application Programming Interface

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.

3.1.1.1. 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.

3.1.1.2. 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.

3.1.1.3. 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*

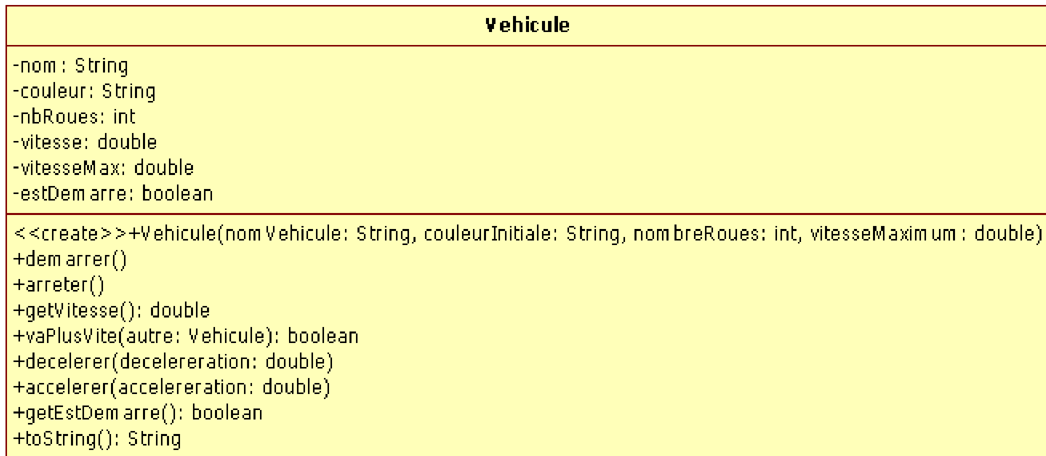
  ```
  monObjet = new Bidon(2,3) ;
  ```

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") ;
```

3.1.2. Vehicule

Figure 2.2. Classe véhicule



Les classes *Vehicule* et *Point* sont disponibles dans le paquetage tp.prof.classes, pour les utiliser vous devez les importer. L'*import* spécifie l'espace de nommage, vous devrez rajouter dans vos fichiers :

```
import tp.prof.classes.Point;
import tp.prof.classes.Vehicule;
```

Le bytecode de Vehicule étant disponible, nous allons l'utiliser pour créer dans tp.etudiant.tests une classe Test-Vehicule. Cette classe ressemblera à :

```
package tp.etudiant.tests;

import tp.prof.classes.Vehicule;

public class TestVehicule {

 /**
  * @param args
  */
 public static void main(String[] args) {
 //au boulot
 }

}
```

Dans cette classe, vous devrez :

- créer une voiture de couleur rouge à quatre roues ne pouvant dépasser 180km/h, l'afficher
- démarrer la voiture, la faire accélérer de 50km/h, afficher la voiture
- la faire accélérer de 40km/h
- faire afficher sa vitesse
- créer un autre véhicule
- démarrer, accélérer, ...
- comparer les vitesses des deux véhicules.
- ...

3.1.3. Point

Figure 2.3. Classe Point

```
Point
-x: int
-y: int
<<create>>+Point(x: int, y: int)
<<create>>+Point()
<<create>>+Point(point: Point)
+getX(): int
+setX(x: int)
+getY(): int
+setY(y: int)
+translater(dx: int, dy: int)
+deplacer(x: int, y: int)
+distanceOrigine(): double
+distance(autrePt: Point): double
+toString(): String
```

Vous disposez d'une API documentant la classe *Point*, dans le paquetage *test* créer une classe *TestPoint*, puis vous :

- Créer un Point p1 initialisé à (0,0), l'afficher.
- Créer un Point p2 initialisé à (1,2), l'afficher.
- Afficher la coordonnée x de p1, puis de p2.
- Modifier la coordonnée y de p2 en y affectant la valeur 3 , afficher p2.
- Translater p2 de 1 en x et de 1 en y, afficher p2.
- Déplacer p1 en (2,10), l'afficher.
- Calculer la distance de p2 à l'origine, l'afficher.
- Calculer la distance entre p1 et p2, l'afficher.
- Affecter p1 à p2, afficher les deux.
- Translater p1 de 2,1, afficher p1 et p2 (surpris(e) ?)

3.1.4. String et StringBuffer

3.1.4.1. 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).

3.1.4.2. 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 ?

3.1.5. 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.

3.1.5.1. 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.
```

3.1.5.2. 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.

3.2. Création d'une classe

Le but de ce TP est de vous faire écrire vos premières classes décrivant un objet (Point et Personne). Une fois ces classes décrivant le fonctionnement d'un objet écrites, les instances de ces classes vont s'utiliser de la même façon que n'importe quel objet de l'API standard.

3.2.1. Point

Figure 2.4. Classe Point

```
Point
-x: int
-y: int
<<create>>+Point(x: int, y: int)
<<create>>+Point()
<<create>>+Point(point: Point)
+getX(): int
+setX(x: int)
+getY(): int
+setY(y: int)
+translater(dx: int, dy: int)
+deplacer(x: int, y: int)
+distanceOrigine(): double
+distance(autrePt: Point): double
+toString(): String
```

Vous avez déjà utilisé la classe *Point* (celle de tp.prof.classes). En vous inspirant de l'exemple du cours, implantez cette classe. La classe `TestPoint.java` du TP précédent doit continuer de fonctionner. Avant de créer votre propre classe *Point*, vous devez modifier TestPoint pour l'import soit

```
import tp.etudiant.classes.Point;
```

et non plus `tp.prof.classes.Point;`.

Vous placerez donc la classe *Point* dans le paquetage *tp.etudiant.classes*.

Important

Vous devez suivre l'API et non pas seulement le diagramme de classe UML.

3.2.2. Personne

Figure 2.5. Classe Personne

```
Personne
-nom: String
-prenom: String
-naiss: Date
<<create>>+Personne(nomIni: String, prenomIni: String, naissIni: Date)
<<create>>+Personne(nom: String, prenom: String)
+getNom(): String
+getPrenom(): String
+getNaissance(): Date
+setNom(newNom: String)
+plusAgee(autre: Personne): boolean
+toString(): String
```

La classe *Personne* décrit (sommairement) une personne . On mémorise trois informations, le nom de la personne, le prénom de la personne et la date de naissance de la personne. On peut comparer l'âge de personnes.

3.2.2.1. Implantation de la classe Personne

Outre de diagramme de classe UML précédent, vous disposez de la documentation de la classe Personne, elle contient l'API de *Personne*. Vous devrez suivre cette API pour implanter la classe *Personne*, dans le paquetage *tp.etudiant.classes*.

3.2.2.2. Réalisation d'une classe de test

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

3.3. 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.

3.3.1. 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.

3.3.1.1. Variables de classes, constantes

Exemple 2.3. Variable de classe

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

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

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

Exemple 2.4. 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 2.5. 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 2.6. 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, ...).

3.3.1.2. Méthodes de classes

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

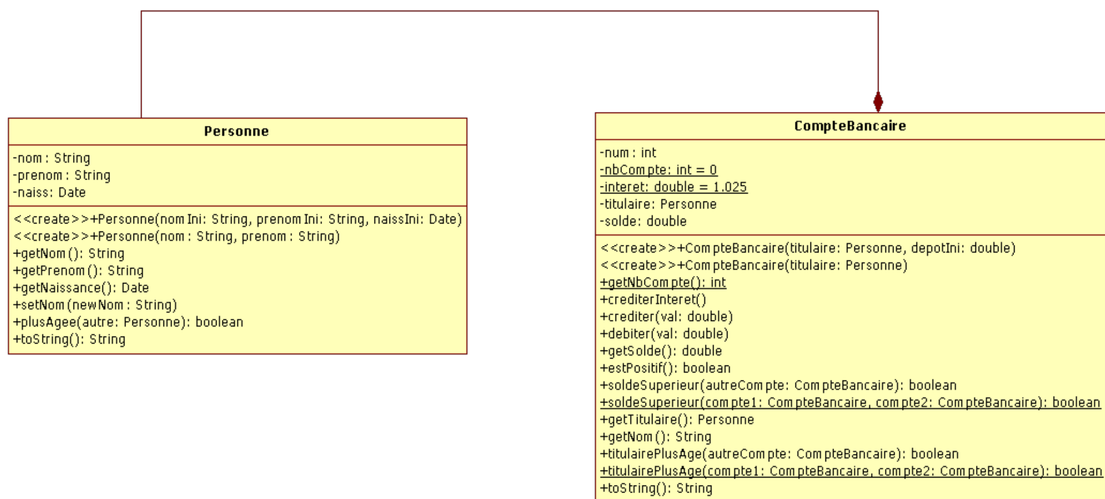
3.3.2. Classe CompteBancaire

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

3.3.2.1. 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 2.6. Comptebancaire



3.3.2.2. Implantation de la classe CompteBancaire

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

3.3.2.3. 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*.

4. Collections

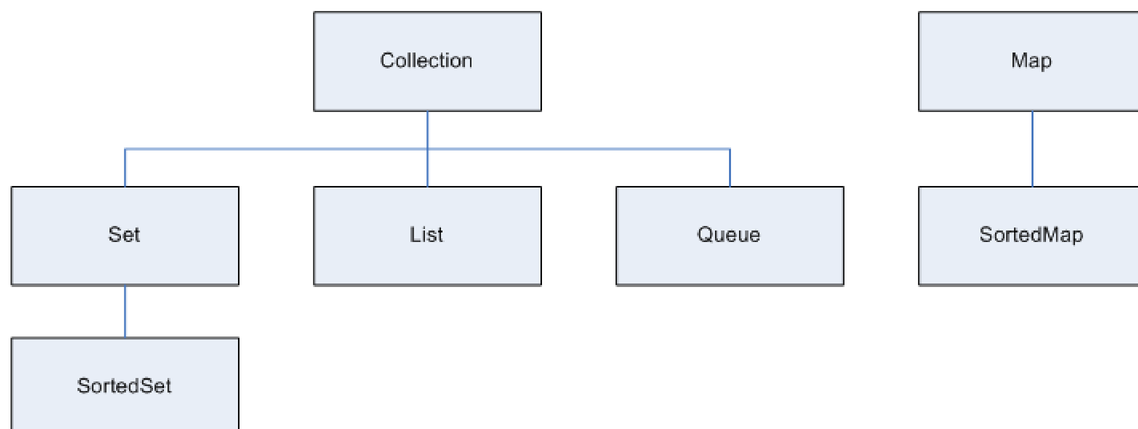
Les tableaux ne sont pas les seules structures de données utilisables en java, il y a aussi les *collections* et les "*maps*" (tableaux associatifs: clef-valeur).

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

4.1. 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[1] « Collection » et s’utilisent « de la même manière ». Le nom des méthodes est fixée.

Figure 2.7. Collections



[1]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.

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,

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

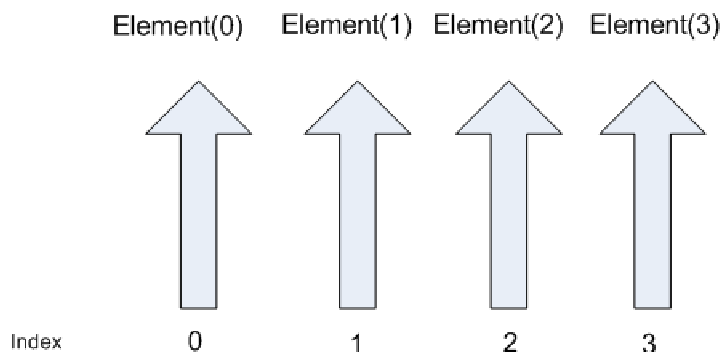
void remove();
    retire le dernier élément accédé.

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

4.2. Listes

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

Figure 2.8. 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 2.7. 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
   */
```

4.3. 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.

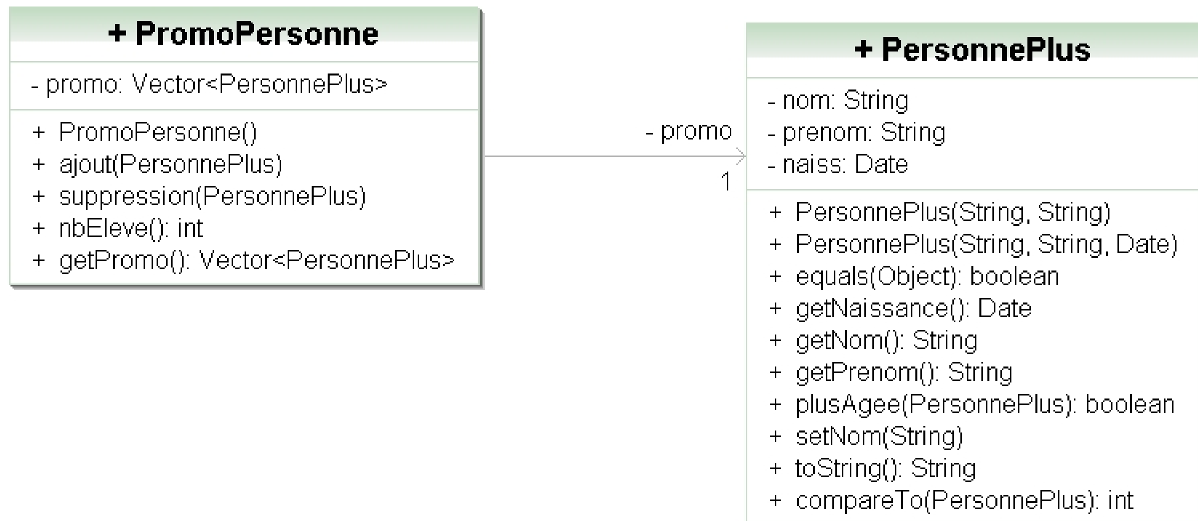
Exemple 2.8. Utilisation d'une *HashTable*

```
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
   */
```

4.4. 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 2.9. Diagramme de classe de la Promotion



4.4.1. 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.

4.4.2. Tableau associatif de personnes

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

5. Récursivité

En informatique et en logique, une fonction ou plus généralement un algorithme qui contient un appel à elle-même est dite récursive. Deux fonctions peuvent s'appeler l'une l'autre, on parle alors de récursivité croisée. Créer une classe utilitaire *Recursivite* qui ne contiendra que des méthodes statiques et coder deux méthodes :

int fac(int n)
: qui implémente une factorielle (fac(n) = n * fac(n-1) et fac(0)=1)

int pgcd(int n, int m)
: qui implémente le calcul du plus grand dénominateur commun (pgcd(n,m) = pgcd(m, n%m) et pgcd(n,0)=n)

Enfin, pour finir pouvez vous expliquer les méthodes car(), cdr() et getTaille() de la classe Liste.

Chapitre 3. 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 3.1. Taquin début du jeux

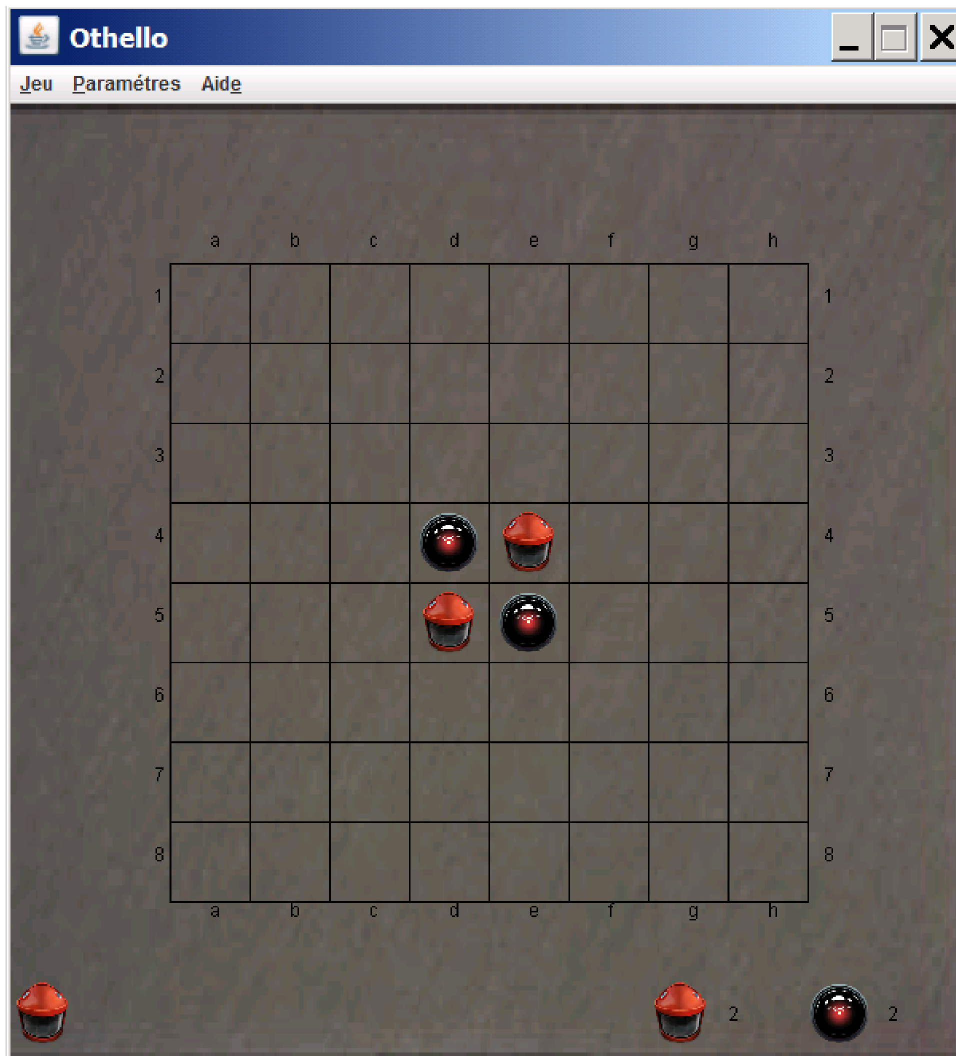
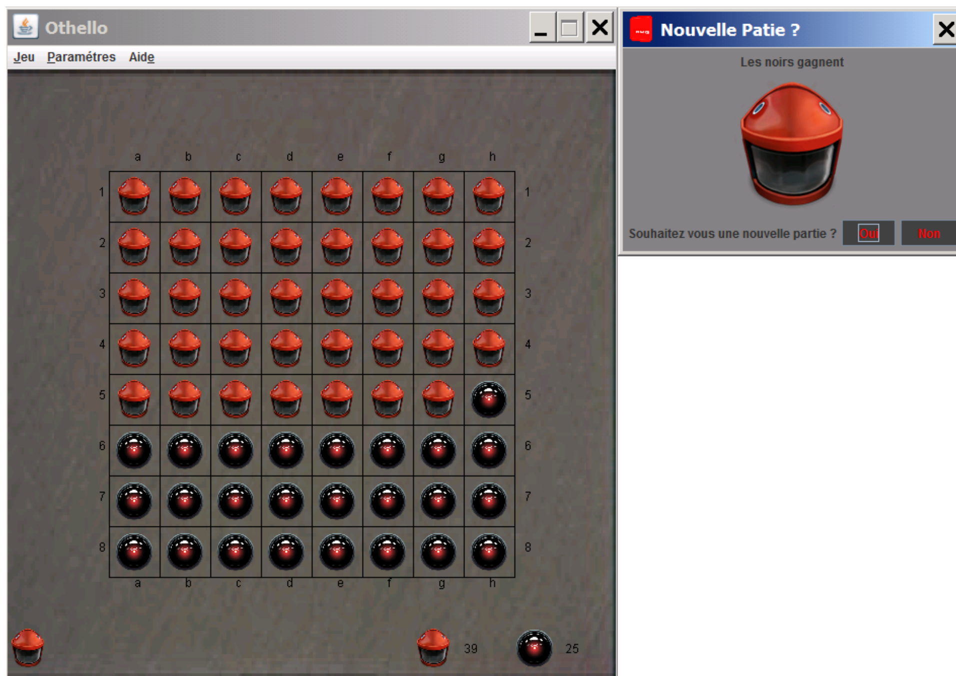


Figure 3.2. Othello partie gagnante



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.

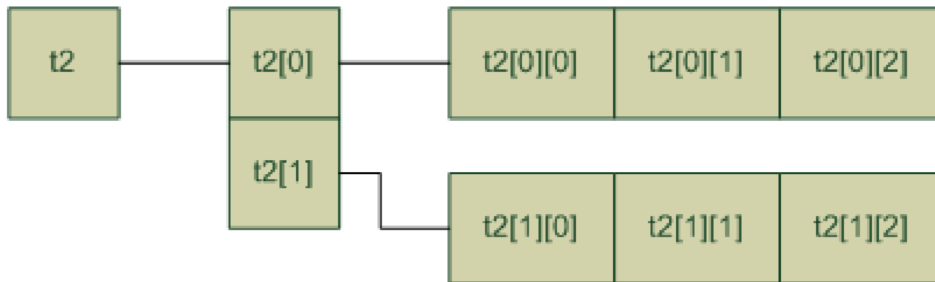
```
int[][] t2;
```

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 3.3. 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 3.4. Représentation logique

(0,0) t2[0][0]	(0,1) t2[0][1]	(0,2) t2[0][2]
(1,0) t2[1][0]	(1,1) t2[1][1]	(1,2) t2[1][2]

(1,0) t2[1][0]	(1,1) t2[1][1]	(1,2) t2[1][2]
(0,0) t2[0][0]	(0,1) t2[0][1]	(0,2) t2[0][2]

(1,2) t2[1][2]	(1,1) t2[1][1]	(1,0) t2[1][0]
(0,2) t2[0][2]	(0,1) t2[0][1]	(0,0) t2[0][0]

(0,2) t2[0][2]	(0,1) t2[0][1]	(0,0) t2[0][0]
(1,2) t2[1][2]	(1,1) t2[1][1]	(1,0) t2[1][0]

(0,0) t2[0][0]	(1,0) t2[1][0]
(0,1) t2[0][1]	(1,1) t2[1][1]
(0,2) t2[0][2]	(1,2) t2[1][2]

(0,2) t2[0][2]	(1,2) t2[1][2]
(0,1) t2[0][1]	(1,1) t2[1][1]
(0,0) t2[0][0]	(1,0) t2[1][0]

(1,2) t2[1][2]	(0,2) t2[0][2]
(1,1) t2[1][1]	(0,1) t2[0][1]
(1,0) t2[1][0]	(0,0) t2[0][0]

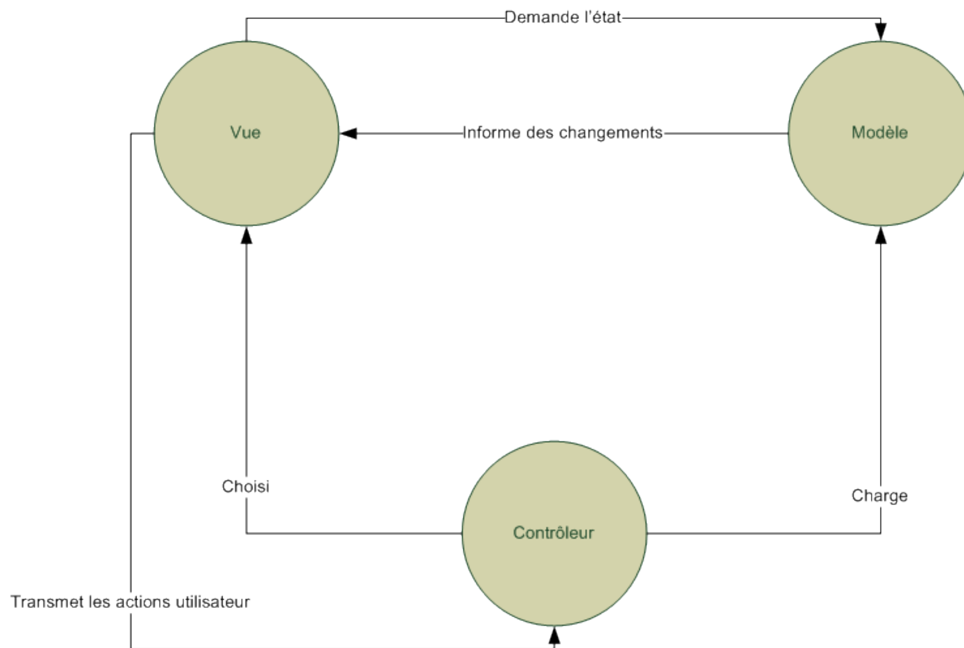
(1,0) t2[1][0]	(0,0) t2[0][0]
(1,1) t2[1][1]	(0,1) t2[0][1]
(1,2) t2[1][2]	(0,2) t2[0][2]

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.

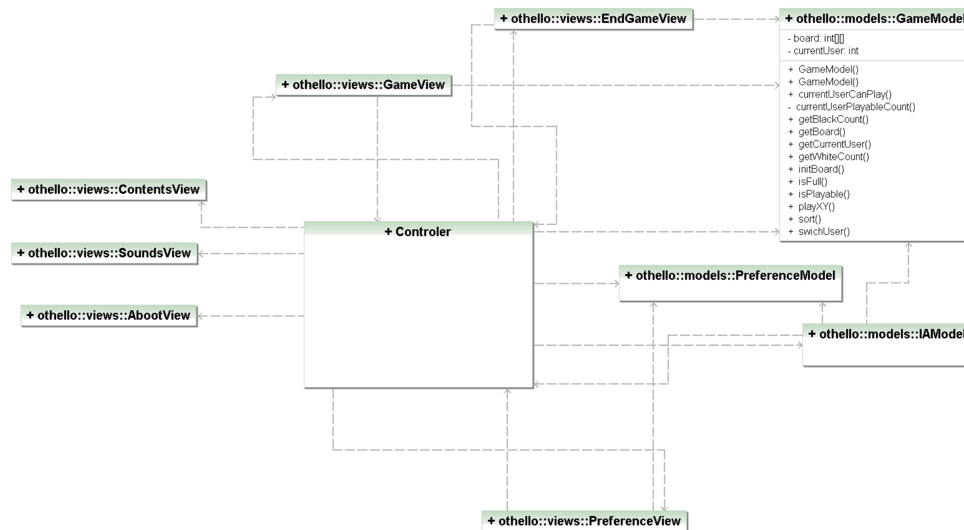
L'application repose sur un modèle qui sépare différents rôle, le patron de conception : Modèle, Vue, Contrôleur (MVC).

Figure 3.5. Modèle Vue Contrôleur



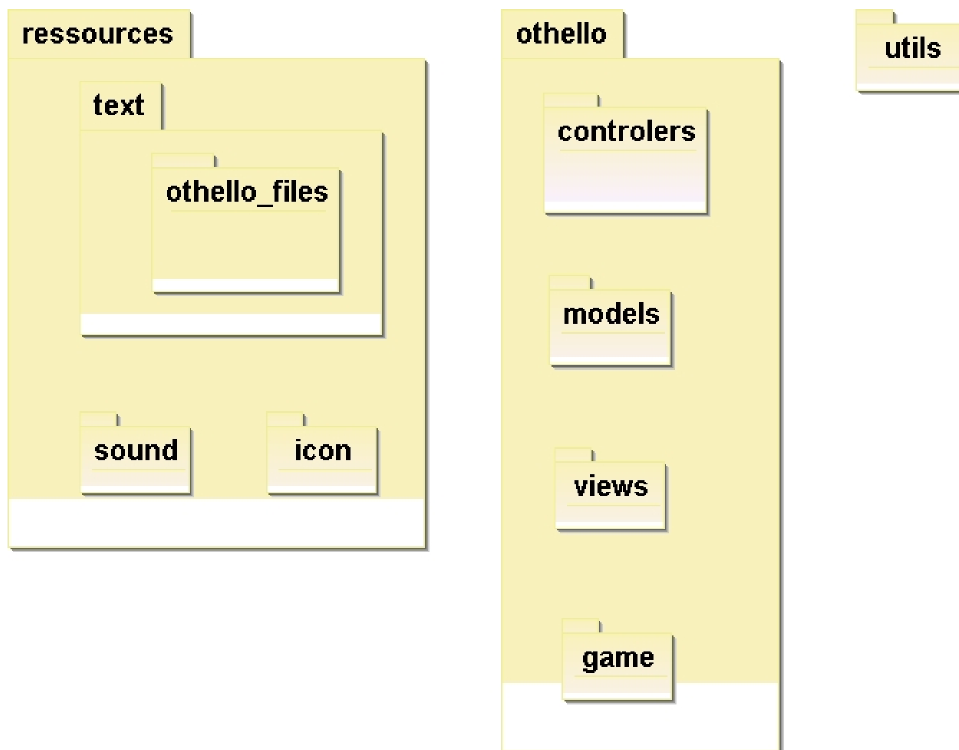
Ne disposant pas encore des outils ne programmation pour réaliser une bonne implémentation du MVC, nous allons en offrir une approche amoindrie. Le diagramme de classe de notre application est le suivant :

Figure 3.6. Diagramme de classe



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

Figure 3.7. 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 30/04/10.

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"

Glossaire