Feuille 1 - Exercices de révision

1 Structures de contrôle.

1. Écrire un programme qui affiche les entiers jusqu'à 100 en remplaçant les multiples de 5 par fizz, les multiples de 7 par buzz et les multiples des deux par fizzbuzz. Utiliser le modulo (%).
2. Écrire un programme qui affiche dans l'ordre inverse les arguments de la ligne de commande, puis le modifier pour afficher en plus à l'envers les arguments eux-mêmes.
3. Écrire un programme qui trie un tableau d'entiers en utilisant la méthode dite de sélection : une première fonction calcule la position du plus petit élément sur une zone, une seconde réalise le tri proprement dit en plaçant les éléments dans l'ordre croissant et une troisième affiche le tableau. La fonction principale remplit aléatoirement un tableau pour tester l'algorithme.

2 Analyse de programmes

1. On considère la définition d'objet suivante :

A

```
1  public class A {
2      public int x;
3      public A(int x){
4          this.x=x;}
5      public int f(){
6          return x+3;
7      }
8      public A g(A a){
9          return new A(2*x+3*a.x);
10     }
11 }
```

   (a) Quel est l'affichage produit par le programme suivant :

TestA

```
1  public class TestA {
2      public static void main(String[] args){
3          A u=new A(1);
4          System.out.println(u.f());
5          A v=new A(4);
6          System.out.println(v.f());
7          A w=u.g(v);
```

```
8           System.out.println(w.f());
9           w.x=3;
10          System.out.println(w.f());
11      }
12  }
```

(b) Dessiner l'état de la mémoire juste après l'exécution de la dernière ligne du programme de la question précédente.

(c) Quel est l'affichage produite par le (morceau de) programme suivant :

```
A u=new A(5);
System.out.println(u);
```

Proposer une méthode d'instance permettant de rendre plus agréable cet affichage.

(d) À la ligne 2 de la classe `A`, on remplace `public` par `private`. Comment faudrait-il modifier le programme de la question 1.a pour que la compilation ne provoque pas d'erreur ?

2. On considère le programme suivant :

B

```
1   public class B {
2       public static int nbB=0;
3       public String n;
4       public B(String n){
5           this.n=n;
6           nbB++;
7       }
8       public String toString(){
9           return "["+n+"]";
10      }
11      public static void nbCons(){
12          System.out.println("nb type B :"+ nbB);
13      }
14
15      public static void main(String[] args){
16          B[] listeB=new B[3];
17          nbCons();
18          listeB[0]=new B("albert");
19          nbCons();
20          listeB[1]=new B("boris");
21          nbCons();
22          for(int i=0; i<B.nbB; i++)
23              System.out.println(listeB[i]);
24      }
25  }
```

(a) Quel est l'affichage produit par son exécution ?

(b) Dessinez l'état de la mémoire juste après l'exécution de la dernière ligne du programme de la question précédente.

(c) On place la fonction `main` dans une autre classe, par exemple `TestB`, comme dans l'exercice précédent. Comment faut-il la modifier pour que la compilation ne provoque pas d'erreur ?

3. On considère le programme suivant :

TestC

```
1  public class TestC {
2      private double var1;
3      private int var2;
4
5      public TestC(){
6          var1=0;
7          var2=0;
8      }
9      public String toString(){
10         return var1+" "+var2;
11     }
12     public void F(TestC t){
13         var1=t.var1+2.5;
14         var2=t.var2+1;
15     }
16     public static void G(int v){
17         v=5;
18     }
19     public static void H(TestC t, double x){
20         x=t.var1;
21         t.var2=3;
22     }
23
24     public static void main(String[] arg){
25         int u=1;
26         double y=3.1;
27         TestC t1= new TestC();
28         TestC t2= new TestC();
29         t1.F(t2);
30         G(u);
31         H(t2,y);
32         System.out.println(u);
33         System.out.println(y);
34         System.out.println(t1);
35         System.out.println(t2);
36     }
37   }
```

(a) Quel est l'affichage produit par son exécution ?

(b) Dessiner l'état de la mémoire à la fin du programme.

3 Objets simples

3.1 Rectangle

Pour programmer une classe `Rectangle` (qui ne sera pas graphique pour simplifier), on commence par définir la classe `PointR2` [1], suivant le squelette ci-dessous, où `x` et `y` représentent les coordonnées dans le plan :

PointR2

```
1 public class PointR2{
2   public double x;
3   public double y;
4   public PointR2(){}
5   public PointR2(double a, double b){}
6   public double distance(PointR2 p){}
7   public boolean equals(Object o){}
8   public String toString(){}
9 }
```

1. Programmer les constructeurs et les méthodes en accord avec la documentation suivante :
   (a) `PointR2()` construit le point origine des coordonnées $(0, 0)$,
   (b) `PointR2(double a, double b)` construit le point de coordonnées (a, b),
   (c) `double distance(PointR2 p)` retourne la distance (euclidienne) entre le point appelant et `p`,
   (d) `boolean equals(Object o)` renvoie `true` si le point appelant et le point paramètre sont les mêmes (i.e. ont les mêmes coordonnées),
   (e) `String toString()` renvoie une représentation sous forme de chaîne de caractères du point appelant : par exemple, "(2.0,3.0)" pour le point de coordonnées $(2, 3)$.
2. On considère maintenant le squelette suivant pour la classe `Rectangle`, un rectangle étant défini par deux points aux extrémités d'une diagonale :

Rectangle

```
1  public class Rectangle{
2    public PointR2 ext1;
3    public PointR2 ext2;
4    public Rectangle(){}
5    public Rectangle(PointR2 p, PointR2 q){}
6    public Rectangle(PointR2 cig, double l, double h){}
7    public double longueur(){}
8    public double hauteur(){}
9    public double perimetre(){}
10   public double surface(){}
```

1. ce nom est choisi pour éviter la confusion avec la classe `Point` qui existe déjà en `Java`

```
11    public boolean contient(PointR2 p){}
12    public Rectangle symetrique(){}
13  }
```

Programmer les constructeurs et les méthodes en accord avec la documentation suivante :

(a) `Rectangle(PointR2 p, PointR2 q)` construit le rectangle dont les points `p` et `q` sont les extrémités d'une diagonale,

(b) `Rectangle()` construit le rectangle réduit à un point (l'origine des coordonnées),

(c) `Rectangle(PointR2 cig, double l, double h)` construit un rectangle dont `cig` est le coin inférieur gauche, `l` la longueur et `h` la hauteur,

(d) `double longueur()` retourne la longueur du rectangle appelant,

(e) `double hauteur()` retourne la hauteur du rectangle appelant,

(f) `double perimetre()` retourne le périmètre du rectangle appelant,

(g) `double surface()` retourne la surface du rectangle appelant,

(h) `boolean contient(PointR2 p)` renvoie `true` si le point paramètre `p` est à l'intérieur du rectangle appelant,

(i) `Rectangle symetrique()` renvoie le rectangle obtenu à partir du rectangle appelant par symétrie par rapport à la première diagonale.

3.2 Date

Afin de permettre l'affichage d'un calendrier, on veut programmer une classe `Date`, avec le squelette suivant :

Date

```
1  public class Date{
2    public int jour;
3    public int mois;
4    public int an;
5    public final static Date mardi010102=new Date(1,1,2002);
6    public final static String[] semaine={"dim", "lundi", "mardi", "merc",
7                                          "jeudi", "vend", "sam"};
8    public Date(int j, int m, int a){}
9    public static boolean biss(int a){}
10   public int nbJoursMois(){}
11   public int nbJoursDepuis1600(){}
12   public int distance(Date d){}
13   public int numJourDeLaSemaine(){}
14   public int compareTo(Date d){}
15   public String toString(){}
16   public void afficheCal(){}
17 }
```

Écrire les méthodes en respectant la documentation suivante :

1. Le constructeur `Date(int j, int m, int a)` construit un objet `Date` représentant la date indiquée par les paramètres (les mois sont numérotés de 1 à 12).
2. La méthode de classe `boolean biss(int a)` renvoie `true` si l'entier `a` correspond à une année bissextile.

   On rappelle les règles suivantes concernant les années bissextiles :
   - le mois de février a 28 jours dans une année ordinaire, 29 dans les années bissextiles.
   - les années bissextiles sont les années divisibles par 4, sauf celles divisibles par 100 mais pas par 400 : 1600, 1980, 1996 étaient des années bissextiles, 1700, 1800, 1900, 1981 ne le sont pas.
3. `int nbJoursMois()` renvoie le nombre de jours du mois correspondant à la date appelante.
4. `int nbJoursDepuis1600()` calcule le nombre de jours écoulés depuis le 1er janvier 1600 (ce jour inclus).
5. `int distance(Date d)` renvoie la distance en nombre de jours entre la date appelante et la date paramètre. On retournera un nombre négatif si `d` précède la date appelante et positif sinon, en utilisant la méthode précédente.
6. `int numJourDeLaSemaine()` retourne le numéro du jour de la semaine correspondant à la date appelante (de 0 pour dimanche à 6 pour samedi). On prendra par exemple comme repère que le 1er janvier 2002 était un mardi.
7. `int compareTo(Date d)` compare deux dates selon le modèle classique : elle retourne 1 si la date appelante est postérieure au paramètre, -1 si elle lui est antérieure et 0 si les dates sont égales.
8. `String toString()` retourne une chaîne représentant la date sous la forme jour/mois/année.
9. `void afficheCal()` affiche le calendrier du mois correspondant à la date appelante.

4 Programmation d'un jeu de dominos

Dans cet exercice, on se propose d'écrire les classes utiles à la programmation d'un jeu de dominos simplifié. Un domino est un petit rectangle en bois divisé en deux parties. Sur chacune est inscrit un numéro entre 0 et 6 : [i, j]. Un domino est donc représenté par deux entiers, domino [i, j] étant le même que le domino [j, i]. Un jeu de dominos complet est donc constitué des 28 dominos suivants :

[6, 6]
[5, 5] [5, 6]
[4, 4][4, 5][4, 6]
[3, 3][3, 4][3, 5][3, 6]
[2, 2][2 ; 3][2, 4][2, 5][2, 6]
[1, 1][1, 2][1, 3][1, 4][1, 5][1, 6]
[0, 0][0, 1][0, 2][0, 3][0, 4][0, 5][0, 6]

4.1 La classe Domino

Cette classe est donnée par son squelette, où les variables d'instance left et right donnent les deux numéros (respectivement gauche et droite) d'un domino :

domino

```
public class domino{
  private int left;
  private int right;
  public Domino (int i, int j){} ;
    public domino(){} ;
    public int getLeft(){} ;
    public int getright{} ;
    public boolean isdomino() ;
    public boolean isDouble(){} ;
    public int totalPoint(){} ;
    public void rotate(){} ;
    public String toString(){} ;
    public int compareTo(Domino d);

}
```

Programmer les constructeurs et les méthodes en accord avec la documentation suivante :

1. Le constructeur `Domino(int i, int j)` construit un domino qui aura comme numéro i à gauche et j à droite. Le constructeur `Domino()` construit le domino [0, 0],
2. Les méthodes " observateurs " `int getLeft()` et `int getRight()` retournent respectivement le numéro de gauche et le numéro de droite du domino appelant.,
3. La méthode boolean `isDomino()` vérifie que le domino appelant en est bien un, c'est-à-dire que ses numéros sont corrects.,
4. La méthode boolean `isDouble()` vérifie si le domino appelant est un double,
5. La méthode int `totalPoint()` retourne la somme des numéros du domino appelant,
6. La méthode void `rotate()` retourne le domino appelant, c'est-à-dire qu'elle transforme le domino appelant [i, j] en le domino [j, i],
7. La méthode `String toString()` renvoie une chaîne de la forme [i ; j],
8. La méthode public `compareTo(Domino d)` permet de comparer un domino à un autre dans le jeu. Elle renvoie : -2 (2) si les coté gauches sont identiques (à l'inverse), -1 (1) si le coté droit du domino appelant coincide avec le cote gauche du domino passé en argument (à l'inverse). Sinon 0. Cette méthode fera appel à la méthode rotate().

4.2 La classe EnsembleDom

On veut ensuite définir des ensembles de dominos, qui seront utilisés pour représenter le jeu initial complet, qui deviendra ensuite la pioche lorsque les dominos auront été distribués aux 2 joueurs.

Le squelette de cette classe est :

EnsembleDom

```
public class EnsembleDom{

    public EnsembleDom(){} ;
    public void add(Domino d){} ;
```

```
5       public boolean isEmpty(){} ;
6       public Domino remove(){} ;
7       public Domino remove(int i){} ;
8       public void consTotal(){};
9       public String toString(){};
10      public int taille() ;
11      public Domino getDom(int i){};
12      public void melanger(){};
13      public string toString(){};
14  }
```

Programmer les méthodes de la classe `EnsembleDom` en respectant la documentation suivante :

1. Le constructeur `EnsembleDom()` produit un ensemble vide de dominos.
2. La méthode `void add(Domino d)` ajoute à l'ensemble le domino en paramètre en fin de liste.
3. La méthode boolean `isEmpty()` renvoie vrai si et seulement si l'ensemble est vide.
4. La méthode `Domino remove()` retourne le dernier domino du tableau et le retire de l'ensemble,

   tandis que `Domino remove(int i)` extrait précisément le domino à la position i.
5. La méthode `void consTotal()` construit l'ensemble complet des 28 dominos.
6.
7. La méthode `String toString()` retourne la chaîne formée par la suite des dominos de l'ensemble. Cette chaîne est ordonnée s'il s'agit du jeu posée sur la table et ne l'est pas dans le cas du jeu d'un joueur.
8. La méthode `int taille()` retourne la taille de l'ensemble.
9. Les méthodes int `getLeft()` et `int getRight()` retournent respectivement la valeur du chiffre à l'extrémité gauche et à l'extrémité droite de l'ensemble.
10. la méthode void `melanger()` permet de mélanger l'ensemble des dominos de la pioche.

4.3 La classe ChainDom

ChaineDom

```
1   import java.util.*; public class ChaineDom {
2       private LinkedList chaine;
3       public ChaineDom(Domino d){}
4       public int left(){}
5       public int right(){}
6       public int jouable(Domino d){}
7       public void addLeft(Domino d){}
8       public void addRight(Domino d){}
9       public String toString(){}
10
11      public static void main(String[ ]args){
12          Domino d1 = new Domino(2,3);
```

```
13          ChaineDom cd = new ChaineDom(d1);
14          Domino d2 = new Domino(3,6);
15          Domino d3 = new Domino(2,5);
16          System.out.println(cd.jouable(d2));
17          System.out.println(cd.jouable(d3));
18          cd.addRight(d2);
19          cd.addLeft(d3);
20          System.out.println(cd);
21      }
22  }
```

Programmer les méthodes de la Classe `ChainDom` en respectant la documentation suivante :

1. Le constructeur `ChaineDom(Domino d)` produit un ensemble contenant le domino d en paramètre.
2. Les méthodes `left()` et `right()` retournent respectivement la valeur du chiffre à l'extrémité gauche et à l'extrémité droite de la chaîne de dominos.
3. La méthode `jouable(Domino d)` permet de comparer le domino d aux deux dominos qui se trouvent aux extrémités de la chaîne. La méthode fait appel à la méthode `compareTo( Domino h)` de la Classe Domino.
4. Les méthodes `addLeft(Domino d)` et `addRight(Domino d)` ajoutent le domino d respectivement à gauche et à droite de la chaîne de dominos.
5. La méthode `toString()` retourne la chaîne formée par la suite des dominos de l'ensemble.

4.4 La classe Joueur

Pour cette classe, il s'agit d'implémenter les méthodes nécessaires aux joueurs pour effectivement jouer une partie. Le jeu de chaque joueur ainsi que son nom sont des variables privées.

Joueur

```
1   public class Joueur {
2       private String name;
3       private EnsembleDom jeu;
4       public Joueur(String s) {}
5       public String getName(){}
6       public boolean gagne(){}
7       public int piocher(EnsembleDom pioche){}
8       public void init(EnsembleDom pioche){}
9       public Domino choixPremierDom(ChaineDom cd){}
10      public String toString(){}
11
12      public static void main(String [] args){
13          EnsembleDom pioche = new EnsembleDom();
14      Joueur j = new Joueur("toto");
15          pioche.consTotal();
16          pioche.melanger();
```

```
17          System.out.println(pioche);
18      j.init(pioche);
19          System.out.println(j);
20      j.piocher(pioche);
21          System.out.println(j);
22          System.out.println(pioche);
23      }
24  }
```

Programmer les méthodes de la classe Joueur en respectant la documentation suivante :

1. Le constructeur `Joueur(String name)` initialise le nom du joueur et crée son jeu.
2. La méthode `String getName()` retourne le nom du joueur.
3. La méthode boolean `gagne()` retourne vrai si le joueur n'a plus de dominos à poser. Dans ce cas, le joueur a gagné.
4. La méthode `piocher(EnsembleDom pioche)` permet d'attribuer au joueur un domino issue de pioche. Il faut tester le cas où elle est vide.
5. La méthode `init(EnsembleDom pioche)` permet d'attribuer au joueur en début de partie sept dominos à partir de pioche.
6. La méthode `choixPremierDom(ChaineDom cd)` permet de choisir un premier domino jouable dans le jeu du joueur.
7. La méthode `toString ()` permet d'afficher les caractéristiques du joueur : son nom et son jeu.

4.5 La classe Partie

La classe Partie met en oeuvre une partie avec deux joueurs en utilisant les méthodes décrites précédemment.

Partie

```
1   public class Partie {
2       private EnsembleDom pioche;
3       private Joueur[] jr;
4       private ChaineDom cd;
5       public Partie(String nomA, String nomB) {}
6       public String toString(){}
7       public int unTour(int i){}
8       public void partieDomino(){}
9
10      public static void main (String [] args){
11          Partie p = new Partie("martine", "robert");
12      System.out.println(p);
13          p.partieDomino();
14      }
15  }
```

Programmer les méthodes de la classe Partie en respectant la documentation suivante :

1. Le constructeur `Partie(String nomA, String nomB)` crée une partie avec deux joueurs nommés nomA et nomB. Le constructeur construit également la pioche qui doit comporter 28 dominos et la mélange. Le tableau jr est également initié au niveau de ce constructeur. Chaque case de ce tableau doit correspondre à un joueur.
2. La méthode `toString ()` permet d'afficher les caractéristiques d'une partie : la pioche, les joueurs et la chaîne de dominos cd sur la table.
3. La méthode `unTour(int i)` permet au joueur i de jouer un tour en essayant de trouver un domino jouable dans son jeu. Si ce n'est pas possible, alors dans ce cas, si la pioche est non vide, le joueur doit retirer un domino à partir de cette dernière. La méthode retourne 2 si le joueur a gagné, 1 s'il a trouvé un domino jouable et 0 sinon.
4. La méthode `partieDomino()` construit une partie de dominos.

5 Sujet de TP

5.1 Fractales

Ecrire une classe Complexe qui permettent d'aditionner et de multiplier des complexes.

L'ensemble de Julia est défini pour les points du plan par l'ensemble des points pour lesquels la suite $z_{n+1} = z_n^2 + c$ ne diverge pas.

z_0 parcourt au début tous les points contenus dans un carré centré sur l'origine, chaque point correspondant à un complexe. On peut par exemple choisir pour $c = -0.05 + i \times 0.7$.

Ecrire un programme qui calcule et affiche un ensemble de Julia.

L'ensemble de Mandelbrot est défini par l'ensemble des points c tels que $z_{n+1} = z_n^2 + c$ ne diverge pas avec $z_0 = 0$.

Modifier le programme pour qu'il affiche l'ensemble de Mandelbrot.

5.2 Jeu de dominos

Programmer les différentes classes du jeu de dominos en suivant les instructions du TD.

Feuille 2 ArrayList et récursivité

1 Méthodes récursives

1.1 Calcul de potentiel

Le potentiel d'un nombre entier n est obtenu de la façon suivante : on effectue le produit de ses chiffres, puis le produit des chiffres du résultat trouvé, et on continue cette opération jusqu'à obtenir un nombre d'un seul chiffre. La longueur de la chaîne ainsi obtenue est appelée le potentiel du nombre n.
Exemple : 5 est le potentiel de 77

$$77 \longrightarrow 49 \longrightarrow 36 \longrightarrow 18 \longrightarrow 8$$

1. Produit des chiffres.
   (a) Donner une formule récursive pour le calcul du produit des chiffres d'un nombre entier (observer que le chiffre des unités de n est obtenu par $n\%10$, tandis que $n/10$ est le nombre contenant tous les chiffres de n sauf celui des unités).
   (b) Donner le domaine de validité de cette formule ainsi que les valeurs hors du domaine.
   (c) Programmer une méthode statique récursive `int prodR(int n)` qui retourne le produit des chiffres du nombre `n`.
   (d) Programmer une méthode statique itérative `int prodI(int n)` qui réalise la même opération.
2. Potentiel.
   (a) Donner une formule récursive pour le calcul du potentiel d'un nombre entier.
   (b) Donner le domaine de validité de cette formule et les valeurs hors du domaine.
   (c) Programmer une méthode statique récursive `int potR(int n)` qui retourne le potentiel du nombre `n`.
   (d) Programmer une méthode statique itérative `int potI(int n)` qui réalise la même opération.
   (e) Programmer une méthode statique `int maxPot(int n)` qui retourne le plus fort potentiel pour les entiers compris (au sens large) entre `0` et `n`.
   (f) Programmer une méthode `main` affichant le(s) nombre(s) inférieur(s) à 1000 (au sens large) de plus fort potentiel.

2 Le tri arborescent.

Le principe de ce tri est de construire un Arbre Binaire de Recherche (A.B.R.) et de l'afficher suivant un parcours infixé.

Nous rappelons qu'un A.B.R. est un arbre Binaire dont chaque sommet :

– est porteur d'une valeur.
– tous les fils droits d'un sommet portent une valeur plus grande que lui.
– tous les fils gauches d'un sommet portent une valeur plus petite que lui.

Une fois un A.B.R. construit, si on l'affiche dans l'ordre infixé **en l'écrasant** (métaphoriquement), on obtient un ensemble trié.

Vous créerez deux versions :
– Une version qui utilisera un tableau *static* dont les éléments seront repérés à l'aide d'indices. Dans ce cas les méthodes seront des méthodes de classe (*static).*
– Une version qui utilisera une classe Abr est dont les éléments seront repérés par de références. Dans ce cas les méthodes seront des méthodes d'instance.

Lors de la programmation, vous vous remémorerez qu'en JAVA, le passage des paramètres se fait par valeur.

2.1 Les méthodes de classes

En utilisant les déclarations suivantes :

```
public class Noeud
  {public int val,g,d;}
class Abr{
    private static Noeud[] TabAbr ;
    public static void inserI(int [] t){}
    public static void inserR(int [] t){}
    private static void inserR(int inser, int racine){}
    public static void AffichR(int racine){}
}
```

Vous programmerez les méthodes décrites ci dessous :
– La méthode `inserI (int[] t)` contsruit l'A.B.R dans le tableau `TabAbr` de manière itérative à partir du tableau d'entier t.
– La méthode `inserR (int[] t)` contsruit l'A.B.R dans le tableau `TabAbr` à partir du tableau d'entier t, en appelant la méthode récursive `inserR(int inser, int racine)`.
– La méthode `AffichR(int racine)` affiche l'A.B.R de racine n° Num dans le tableau TabAbr. `TabAbr` de manière récursive de manière à ce que les éléments apparaissent triés.

Par exemple si nous avons la déclaration suivante : `int Tab[] = {20,10,45,30,5,15}` et que l'on appelle `inserR ( Tab)`, nous obtiendrons le tableau `TabAbr` décrit ci dessous, qui après l'appel de `AffichR(0)` affichera le résultat : `5,10,15,20,30,40`.

Val	g	d
20	1	2
10	4	5
45	3	0
30	0	0

Tableau : TabAbr

Dans le tableau `TabAbr` la colonne g indique l'indice de l'élément situé à gauche (donc plus petit) de l'élement courant, la colonne d l'indice de l'élément situé à droite (donc plus grand) de l'élement courant. Lorsque cette valeur est nulle elle indique qu'il n'y a pas d'élément présent puisque l'élément d'indice 0 est par construction la racine de l'arbre.

Vous écrirez ensuite une méthode `public void main(String [] param)` qui permet de tester les méthodes précédentes.

2.2 Les méthodes d'instances

En utilisant les déclarations suivantes :

```
public class Abr {
  private int val;
  private Abr g,d;
  public  Abr (int v){};
  public  Abr getNoeudD(){};
  public  Abr getNoeudG(){};
  public  void affichR(){ };
  public  void ajoutR(int []v ){
  private void ajoutR(Abr n ) {
  public  void ajoutI(int []v ){};
  private void ajoutI(Abr n){}
  public  void affichI(){};
};
```

Bâtir deux méthodes :

- `public void ajoutR(int []v);`
- `private void ajoutR(Abr n);`

qui permettent de batir un Abr à partir d'un tableau d'entiers de manière récursive.

Bâtir deux méthodes :

- `public void ajoutI(int []v);`
- `private void ajoutI(Abr n);`

qui permettent de batir un Abr à partir d'un tableau d'entiers de manière itérative. Que pouvez vous dire sur la comparaison de la programmation itérative et de la programmation récursive ?

Bâtir la méthode :

- `public void affichR();`

qui permet d'afficher un `Abr` de manière récursive.

Bâtir la méthode :

- `public void affichI();`

qui permet d'afficher un `Abr` de manière itérative. (La programation de cette dernière méthode demande quelques réflexions.)

Que pouvez vous dire sur la comparaison de la programmation itérative et la programmation récursive ? Comment écrire la méthode *toString* ?

Vous écrirez enfin une méthode `public void main(String [] param)` qui permet de tester les méthodes précédentes.

3 Programmation d'ensembles d'entiers

Cet exercice reprend une partie (légèrement modifiée) du partiel de mai 2006, il est centré sur les ArrayList

On souhaite manipuler des ensembles d'entiers ayant un nombre quelconque d'éléments, avec les opérations habituelles de manipulation des ensembles : union, intersection, test d'inclusion, etc. Pour cela, on définit une classe `EnsA`, où un ensemble est représenté par un objet de la classe `ArrayList`. Pour que les éléments (entiers) puissent être considérés comme des objets, ils seront représentés par le type `Integer` (avec une majuscule) qui est le nom d'une classe Enveloppe.

Nous utilisons explicitement des éléments génériques. La déclaration se fait de la manière suivante `ArrayList<Integer> liste;` ou `<Integer>` indique le type de l'élément qui sera contenu dans `ArrayList`. Cette différence se traduit essentiellement par la génération automatique de *"cast"* qui compense cette déclaration aux formes un peu barbare pour le non-initié par une utilisation allégée, ou le type des éléments récupérés sera automatiquement `<Integer>` (dans notre exemple) . Le squelette de la classe `EnsA` est le suivant :

```
import java.util.*;
public class EnsA {
    private ArrayList<Integer> liste;
    public EnsA(){}
    public EnsA(int a){}
    public EnsA(EnsA e){}
    public boolean isEmpty(){}
    public int cardinal(){}
    public boolean singleton(){}
    public String toString(){}
    public boolean appartient(int a){}
    public void add(int a){}
    private void remove(int a){}
    private int getE(int i){}
    public EnsA union(EnsA e){}
    public EnsA inter(EnsA e){}
    public EnsA diff(EnsA e){}
    public boolean inclus(EnsA e){}
    public boolean equals(Object o){}
}
```

On rappelle que la méthode `int intValue()` convertit l'objet appelant en un élément de type `int` (lorsque c'est possible).

Rappel des méthodes principales de la classe `ArrayList` :

```
public class ArrayList<T> {  // T est le type des objets qui seront contenu dans la liste

    //construit une nouvelle liste vide (T est le type des éléments contenus dans la liste
    public ArrayList()<T>;

    //renvoie la taille (nombre d'éléments) de la liste
    public int size();
    //ajout de l'élément o à la fin de la liste et mise à jour de la taille
    public boolean add(T o);    // T est la type d'objet défini au moment de la déclaration
    //suppression de la première occurrence de o de la liste et mise à jour de la taille
    public boolean remove(Object o);
    //renvoie l'élément d'indice i de la liste
    public Object get(int i);
    //renvoie l'indice de l'élément o dans la liste
    public int indexOf(Object o);
}
```

1. Programmer les constructeurs et les méthodes de la classe `EnsA` en respectant les contraintes suivantes.
   (a) Le constructeur `EnsA()` consruit l'ensemble vide (une liste vide).
   Le constructeur `EnsA(int a)` construit le singleton {`a`}
   Le constructeur `EnsA(EnsA e)` construit un **nouvel** ensemble en copiant tous les éléments de l'ensemble `e`.
   (b) La méthode `int cardinal()` renvoie le cardinal (nombre d'éléments) de l'ensemble appelant. La méthode `boolean isEmpty()` renvoie `true` si et seulement si l'ensemble appelant est vide. La méthode `boolean singleton()` renvoie `true` si et seulement si l'ensemble appelant est un singleton.
   (c) La méthode `String toString()` retourne une chaîne de caractères représentant l'ensemble.
   (d) La méthode `boolean appartient(int a)` renvoie `true` si et seulement si l'entier `a` appartient à l'ensemble appelant.
   (e) La méthode `void add(int a)` ajoute l'élément `a` à l'ensemble appelant s'il n'y est pas. Si cet élément est déjà présent, la méthode ne fait rien.
   (f) La méthode `void remove(int a)` retire l'élément `a` de l'ensemble appelant (si cet élément n'y est pas, la méthode ne fait rien).
   (g) La méthode `EnsA inter(EnsA e)` retourne un nouvel ensemble formé par l'intersection de l'ensemble appelant et de l'ensemble `e` en paramètre. On rappelle que l'intersection de deux ensembles contient exactement les éléments présents à la fois dans ces deux ensembles.
   (h) La méthode `EnsA union(EnsA e)` retourne un nouvel ensemble formé par la réunion de l'ensemble appelant et de l'ensemble `e` en paramètre. On rappelle que la réunion de deux ensembles contient exactement les éléments présents dans l'un ou l'autre de ces deux ensembles.

(i) La méthode `EnsA diff(EnsA e)` retourne un nouvel ensemble formé par la différence de l'ensemble appelant et de l'ensemble `e` en paramètre. On rappelle que la différence de deux ensembles `f` et `e`, notée habituellement $\texttt{f} \setminus \texttt{e}$, contient exactement les éléments de `f` qui ne sont pas dans `e`. L'appel `g=f.diff(e)` affecte donc à `g` l'ensemble $\texttt{f} \setminus \texttt{e}$.

(j) La méthode `boolean inclus(EnsA e)` retourne `true` si et seulement si l'ensemble appelant est contenu dans l'ensemble `e` en paramètre.

(k) La méthode `boolean equals(Object o)` retourne `true` si et seulement si l'ensemble appelant a exactement les mêmes éléments que le paramètre `o` (qui doit donc être un ensemble).

2. Donner le résultat de l'exécution du programme suivant :

```
public class EnsTest {
    public static void main(String[] arg){
EnsA e1= new EnsA(3);
System.out.println(e1);
System.out.println(e1.appartient(4));
e1=e1.union(new EnsA(3));
System.out.println(e1);
e1=e1.union(new EnsA(4));
  System.out.println(e1.cardinal());
System.out.println(e1.singleton());
System.out.println(e1);
EnsA e2= new EnsA(8);
EnsA e12 = e2.union(e1);
System.out.println(e1.inclus(e12));
System.out.println(e12);
EnsA e3= new EnsA(e12);
System.out.println(e3.equals(e12));
System.out.println(e3);
e3=e3.diff(new EnsA(5));
System.out.println(e3);
e3=e3.diff(new EnsA(3));
System.out.println(e3);
e3=e3.union(new EnsA(9));
System.out.println(e3);
EnsA e4= e3.inter(e12);
System.out.println(e4);
EnsA e5=e3.diff(e4);
System.out.println(e5);
e5=e3.diff(e12);
System.out.println(e5);
System.out.println(e3);
System.out.println(e3.equals(e5));
e4.add(9);
System.out.println(e4);
System.out.println(e3);
System.out.println(e3.equals(e4));
    }
}
```

4 T.P.

1. Écrire un programme qui modifie un tableau d'entiers en rangeant ses éléments dans l'ordre inverse de l'ordre initial. Donner une version récursive et une version itérative.

2. On souhaite faire un programme `anagrammes` qui cherche à afficher tous les anagrammes d'un tableau de caracteres.

   Les anagrammes d'un mot sont tous les mots contenant les mêmes lettres dans tous les ordres possibles (i.e. toutes les permutations des lettres). Par exemple les anagrammes de "abc" sont "abc","acb","bac","bca","cba","cab".

   On écrira une procédure qui renvoie un arraylist contenant tous les anagrammes d'un tableau de caractères passé en paramètre. On remarque que pour engendrer tous les anagrammes, il suffit de permuter la première lettre du tableau avec toutes les autres, puis d'afficher tous les anagrammes commençant par la lettre permutée. Ecrire un classe TabAnagramme qui contiendra

   ```
   public class TabAnagramme  {
   private Arraylist<String> tab
   TabAnagramme(){}
   public Arraylist<String> gettab(){}
   public void anagrammes (Char[] mot, int indice){}
   }
   ```

   `Tabanagramme()` initalise `tab`, `gettab()` renvoie `tab`, `anagrammes` prend en argument :
   – `mot` : une chaine dont on veut afficher tous les anagrammes ;
   – `indice` : un entier qui indique le nombre de lettres déjà permutées.
   et qui ajoute à tab tous les anagramme de la chaîne de charactères `mot`.

3. Comparaison de la recherche simple et de la recherche dichotomique dans un tableau trié. Dans une classe `Cherche`, vous programmerez :
   – une méthode `static int ChercheSimple(int [] TabTri,int val,int deb,int fin)` qui retourne le place ou se trouve un entier de valeur `val` dans la partie du tableau trié, `TabTri` qui commence a l'emplacement `deb` et se termine à l'emplacement `fin`. Si l'élément n'est pas présent retourner -1.
   – une méthode `static int ChercheDichoRec(int []TabTri,int val,int deb,int fin)` utilisant la recherche dichotomique de manière récursive.
   – une méthode `int ChercheDichoIt(int [] TabTri,int val,int deb,int fin)` utilisant la recherche dichotomique de manière itérative.
   Vous utiliserez ces méthodes sur un tableau généré au hasard de 10000 entiers, pour lequel vous testerez la présence de 10 entiers. Pour chaque méthode vous indiquerez combien de comparaisons ont été nécessaire avant de trouver la solution.

4. Programmez le tri arborescent "statique" de manière récursive puis itérative.

5. Programmez la classe `EnsA`

   A cette occasion vous programmerez aussi pour chaque classe une méthode `String toString()`. Vous crerez une méthode `public static void main(String []p)`, qui mettra en valeur le fonctionnement des classes et méthodes écrites précédement.

Feuille 3 - Les interfaces

1 Analyse de programmes

1. On considère l'interface suivante :

Bidule
```
public interface Bidule {
        public double map (double x);

        public int combine (int x, int y);

        public void put (String s) ;
}
```

Pour chacune des classes suivantes, indiquer si la classe implante correctement l'interface en précisant, le cas échéant la ou les source(s) d'erreur(s).

(a) BiduleImp1 :

BiduleImp1
```
public class BiduleImp1 implements Bidule {
        private String save ;
        public double map (double x) {
                return 2 * x ;
        }
        public int combine (int x, int y) {
                return x + y ;
        }
        public void set (String s) {
                save = s ;
        }
}
```

(b) BiduleImp2 :

BiduleImp2
```
public class BiduleImp2 implements Bidule {
        private String save ;
        public double map (double x) {
                return Math.sqrt (x) ;
        }
        public int combine (int x, int y) {
                return x/y ;
        }
        public void put (String chaine) {
                save = chaine ;
        }
```

```
12          public String get () {
13                  return save ;
14          }
15  }
```

(c) BiduleImp3 :

BiduleImp3

```
1   public class BiduleImp3 implements Bidule {
2           public double map (double x) {
3                   return  x/2 ;
4           }
5           public int combine (int x, int y) {
6                   return ((double)x)/y ;
7           }
8           public void put (String s) {
9           }
10  }
```

(d) BiduleImp4 :

BiduleImp4

```
1   public class BiduleImp4 implements Bidule {
2           public double map (double x, double y) {
3                   return x + y ;
4           }
5           public int combine (int x, int y) {
6                   return x - y ;
7           }
8           public void put (String s) {
9                   System.out.println (s) ;
10          }
11  }
```

2. On considère les trois interfaces suivantes :

InterA

```
1   public interface InterA {
2           public double f (double x) ;
3   }
```

InterB

```
1   public interface InterB extends InterA {
2           public int g (int x) ;
3   }
```

InterC

```
1   public interface InterC extends InterA {
2           public String h (String s) ;
3   }
```

Soit la classe ImplABC suivante :

ImplABC

```
1   public class ImplABC implements InterA, InterB, InterC {
2           public double f (double x) {
```

```
3                   return 2 * x ;
4           }
5           public int g (int x) {
6                   return x/2 ;
7           }
8           public String h (String s) {
9                   return "'" + s + "'" ;
10          }
11  }
```

On considère le problème test suivant :

TestImplABC

```
1   public class TestImplABC {
2      public static void main (String [] args) {
3           ImplABC obj = new ImplABC () ;
4           System.out.println (obj.f (-1.5)) ;
5           System.out.println (obj.g (5)) ;
6           System.out.println (obj.h ("gnark")) ;
7           InterA a = obj ;
8           System.out.println (a.f (2.34)) ;
9           System.out.println (a.g (-5)) ;
10          System.out.println (a.h ("pouic")) ;
11          InterB b = obj ;
12          System.out.println (b.f (-4.37)) ;
13          System.out.println (b.g (31)) ;
14          System.out.println (b.h ("titi")) ;
15          InterC c = obj ;
16          System.out.println (c.f (13)) ;
17          System.out.println (c.g (13)) ;
18          System.out.println (c.h ("toto")) ;
19          b = c ;
20          a = c ;
21          c = a ;
22          c = b ;
23     }
24  }
```

(a) Indiquer les lignes qui sont refusées à la compilation (en précisant pourquoi).

(b) En supposant que ces lignes sont supprimées, indiquer l'affichage produit par le programme.

2 Utilisation des interfaces

1. Réécrire la méthode de tri par sélection de la Feuille de TD 1 en utilisant l'interface `Comparable` définie dans l'API Java. Écrire par la suite une fonction principale qui permet de trier un tableau d'entier et un tableau de chaîne de caractères. On rappelle l'interface `Comparable` :

Echantillon
```
public interface Comparable {
    public int compareTo(Object o);
}
```

L'appel `a.compareTo(b)` renvoie un entier strictement négatif si `a` est strictement plus petit que `b`, strictement positif si `a` est strictement plus grand que `b` et enfin 0 si `a` et `b` sont égaux.

2. Étant donnée l'interface `Figure` ci-après, donner une programmation des classes `Cercle`, `Carre` et `Rectangle` qui l'implémentent. L'ordre naturel qu'on implémentera se basera sur le périmètre (*i.e.* un carré est plus petit q'un rectangle si son périmètre est plus petit que celui du rectangle et inversement). La méthode `toString` doit renvoyer une chaîne de caractère qui décrit l'instance appelante en précisant son type (carré, rectangle, cercle) et ses caractéristiques (périmètre et aire).

Figure
```
public interface Figure extends Comparable {
        public double perimetre () ;
        public double aire () ;
        public String toString () ;
}
```

3. On donne l'interface suivante, destinée à représenter des ensembles d'entiers positifs :

Ensemble
```
public interface Ensemble {
        public int cardinal ();
        public boolean estVide();
        public void ajouter(int n);
        public boolean appartient(int n);
        public Ensemble union(Ensemble G);
        public Ensemble intersection(Ensemble G);
        public boolean inclus(Ensemble G);
        public String toString();
}
```

   – Donner une programmation de cette interface basée sur des listes d'entiers.

3 Programmation de grands entiers

On souhaite pouvoir manipuler des entiers naturels ayant un nombre quelconque de chiffres. Pour cela, on définit une classe `GrandEntier`, où un entier est représenté dans un objet de la classe `ArrayList`, par la suite de ses chiffres. Les chiffres successifs du nombre seront **obligatoirement** rangés dans la liste dans l'ordre suivant : d'abord le chiffre des unités à l'indice 0, puis le chiffre des dizaines 'a l'indice 1, etc. En effet, cet ordre évitera des manipulations maladroites par la suite, par exemple pour calculer l'addition de deux entiers où les nombres doivent être alignés sur les chiffres respectifs des unités. Par exemple, le nombre 4 492 105 337 est représenté par la suite : [7, 3, 3, 5, 0, 1, 2, 9, 4, 4], avec 7 à l'indice 0 de la liste, 3 aux indices 1 et 2, etc. Enfin, pour que ces chiffres puissent être considérés comme des objets, ils seront représentés par le type `Integer` (avec une majuscule).

Le squelette de la classe `GrandEntier` est le suivant :

```
import java.util.*;
public class GrandEntier implements Comparable {
    private ArrayList<Integer> liste;
    public GrandEntier(){}
    public GrandEntier(int n){}
    public void randomize(int n){}
    public String toString(){}
    public boolean equals(Object o){}
    public int compareTo(Object o){}
}
```

Programmer les constructeurs et les méthodes de la classe `GrandEntier` en respectant la documentation suivante.

1. Le constructeur `GrandEntier()` consruit l'entier 0, qui correspond à une liste vide.
2. Le constructeur `GrandEntier(int n)` construit la représentation de l'entier `n` donné en paramètres. Pour écrire cette méthode, on ajoutera à la liste les différents chiffres de la représentation décimale de `n` dans l'ordre spécifié plus haut : observer que le chiffre des unités de n est obtenu par $n\%10$, tandis que $n/10$ est le nombre contenant tous les chiffres de n sauf celui des unités.
3. La méthode `void randomize(int n)` tire au hasard les `n` chiffres d'un grand entier.
4. La méthode `String toString()` retourne une chaîne de caractères correspondant au grand entier représenté. Attention, cette chaîne doit être conforme à l'écriture habituelle, donc une application brutale de la méthode `String toString()` de la classe `ArrayList` ne convient pas puisqu'elle produit une chaîne contenant la suite des éléments de la liste dans l'ordre croissant des indices.
5. La méthode `boolean equals(Object o)` retourne `true` si et seulement si le grand entier appelant est égal à celui donné en paramètre. Il s'agit bien sûr de l'égalité au sens habituel sur les entiers : même nombre de chiffres et mêmes chiffres aux mêmes positions.
6. La méthode `int compareTo(Object o)` compare deux grands entiers.
7. Utiliser le tri par sélection pour trier un tableau de grands entiers qu'on aura tirés au hasard.

4 TP

1. Programmer le tri par sélection de `Comparable`.
2. Programmer une classe pour les ensembles d'entiers qui implante l'interface `Ensemble`.
3. Le jeu du virus est une simplification du jeu vidéo *Attaxx*. Le plateau de jeu est une grille carrée qui contient au début deux pions noirs et deux pions blancs placés sur des coins opposés comme sur le plateau de la figure 1. Un coup consiste à placer un pion de sa couleur sur une des huit cases voisines d'un pion déjà posé. La case doit être vide pour qu'on puisse poser un pion dessus. De plus après avoir posé le pion, tous les pions adverses sur les cases voisines de la case du pion posé changent de couleur et deviennent

de la couleur du pion posé. Un exemple de coup est donné dans la figure 2. Le jeu se termine lorsque le plateau de jeu est rempli. Le gagnant est celui qui a le plus de pions sur le plateau. Le jeu du virus se joue habituellement sur un damier 7×7.

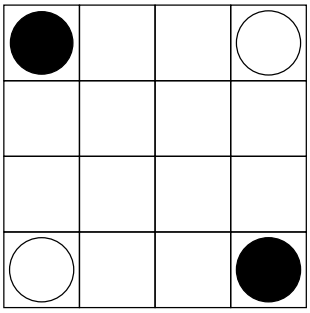

FIGURE 1 – La position de départ au jeu du virus 4x4.

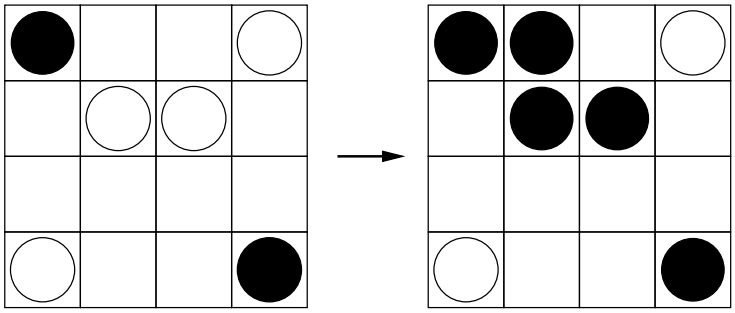

FIGURE 2 – Un coup noir au jeu du virus 4x4.

4. Écrire un programme qui représente un jeu du virus et qui comprend une fonction d'évaluation pour ce jeu. L'évaluation d'une position est égale au nombre de pions noirs moins le nombre de pions blancs. Écrire une intelligence artificielle pour le jeu du virus qui choisit le coup noir qui amène à la position ayant la meilleure évaluation. On incluera dans un coup son évaluation et on implantera l'interface Comparable pour les coups afin de comparer les évaluations et de choisir de jouer le coup qui a la meilleure évaluation.
5. Programmer la classe `GrandEntier` et utiliser le tri par sélection pour trier un tableau de grands entiers.

Feuille 4 -Quelques tris.

Les interfaces `Comparable` et `Comparator`

1 L'interface comparable

On rappelle l'interface Comparable :

Comparable

```
public interface Comparable
  {
    public int compareTo(Object o);
  }
```

L'appel a.compareTo(b) renvoie un entier strictement négatif si a est strictement plus petit que b, strictement positif si a est strictement plus grand que b, et nul s'ils sont égaux.

2 Tri à bulle

Écrire un programme qui ordonne un tableau de Comparable en utilisant la méthode dite de tri à bulle. Le principe est de parcourir le tableau en comparant deux éléments consécutifs que l'on échange au besoin. On parcourt le tableau autant de fois que nécessaire pour que le tableau soit trié. On peut s'éviter les passages inutiles à l'aide d'un booléen qui est mis à true au début de chaque passage, et qui sera positionné à false si un échange a lieu dans la boucle interne.

BULLE

```
public class Bulle
  {
    public static void sort(Comparable[] tab) {}
  }
```

3 Tri par insertion

Écrire un programme qui ordonne un tableau de Comparable en utilisant la méthode dite de tri par insertion. Le principe est le suivant : comme pour trier des cartes, on parcourt le tableau et on insère successivement les éléments dans la partie déjà triée du tableau. On commence par chercher l'indice où l'on doit insérer l'élement qui suit la partie déjà triée du tableau, puis on l'insère en décalant d'une case les élements qui le suivent.

Insertion

```
  public class Insertion
  {
     public static void sort(Comparable[] tab) {}
  }
```

4 Une classe de comparables

Écrire une classe `Personne` implémentant l'interface `Comparable`. Une Personne est caractérisée par son nom, son prénom, et son âge. Une `Personne` sera plus grande qu'une autre, si son âge est plus grand que celui de l'autre. Écrire un constructeur prenant le nom, le prénom et l'âge en paramètres ainsi qu'une méthode `toString()` qui affiche le nom et le prénom de la `Personne`.

Classe Personne

```
1  public class Personne implements Comparable {
2    private String nom;
3    private String prenom;
4    private int age;
5
6    public Personne(String nom, String prenom, int age) {
7      this.nom = nom;
8      this.prenom = prenom;
9      this.age = age;
10   }
11
12   public String toString() {
13     return "{" + nom + " " + prenom +" "+ age + "}\n";
14   }
15
16   public int compareTo(Object o) {
17     Personne p = (Personne)o;
18     return age - p.age;
19   }
20 }
```

5 Utilisation des tris

Écrire une classe `TesteTriComparables` afin de tester les tris programmés précédemment. On suppose que la méthode `desPersonnes(int taille)` renvoie un tableau de personnes et n'est donc pas à programmer.

La méthode `randInteger(int taille)` renvoie un tableau d'objets Integer générés aléatoirement de dimension taille. La méthode `tabVersChaine(Object[] tab)` renvoie une chaine de caractères représentant le tableau passé en argument. La méthode `main(String[] args)` crée un tableau de Personne et un tableau d'Integer, les affiche, les trie, puis affiche le résultat.

TesteTriComparables

```
import java.util.*;
public class TesteTriComparables {
public static Personne[] desPersonnes(int taille) {
    Personne[] tab = new Personne[taille];
    for(int i = 0; i < taille; i++) {
      tab[i] = new Personne("nom"+i, "prenom"+i, (int)(Math.random()*100));
    }
```

```
    return tab;
  }
public static Integer[] randIntegers(int taille) {}
public static String tabVersChaine(Object[] tab) {}
public static void main(String[] args) {}
}
```

6 Comparator

On rappelle l'interface Comparator :

Comparator

```
public interface Comparator
 {
    public int compare(Object o1, Object o2);
    public boolean equals(Object obj);
 }
```

L'appel `compare(a, b)` renvoie un entier strictement négatif si a est strictement plus petit que b, strictement positif si a est strictement plus grand que b, et nul s'ils sont égaux.

7 QuickSort

Écrire un programme qui trie un tableau d'objets en utilisant l'interface `Comparator` et la méthode dite `QuickSort`. Son principe est le suivant :

- On choisit un élément du tableau "au hasard" le pivot.
- On partitionne le tableau en 2 parties à l'aide de la procèdure `partition` : les éléments supérieur au pivot, les éléments inférieurs au pivot, puis l'on place le pivot entre ces deux parties.
- On recommence cette opération récursivement sur chacune des parties obtenues. A la fin le tableau initial est trié.

L'algorithme se décompose en trois phases :

- choix d'un élément pivot. Pour cet exercice, le pivot sera l'élément le plus à gauche du tableau ;
- création des partitions suivant la valeur du pivot à l'aide de la procédure `partition` ;
- appels récursifs pour trier chacune des partitions.

7.1 Partitionnement

Soit un tableau de nombre contenant un nombre V1. Il s'agit de répartir les éléments du tableau en deux parties : les nombres plus grand que V1, les nombres inférieurs à V1, V1 faisant parti des éléments initiaux du tableau. V1 est appelé le **Pivot**, il est choisi au hasard par exemple à l'extrémité droite du tableau. La manière de procéder est la suivante :
Nous avons 3 indices :

- l'indice gauche ou IG
- l'indice droite ou ID
- l'indice du pivot ou IP

Dans l'exemple suivant, la première ligne du tableau représente les valeur à partitionner. Le numéro en italique situé dans la deuxième ligne répresente les indices des cases de la première ligne.

Au départ on place IP sur l'élément droite du tableau, IG sur l'élément gauche, et ID immédiatement à gauche de IP. IP pointe sur l'élement d'indice 9 dont la valeur est 8. Le partitionenment va consister à mettre dans la partie gauche du tableau tous les nombres inférieurs à 8. Dans la partie droite tous les nombres supérieurs à 8, et entre le 2 parties, le 8.

5	3	10	6	15	7	16	4	8
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Le partitionnement va consister à

- Faire avancer IG vers la droite en s'arrêtant sur tous les nombres plus grands que le pivot.
- Faire avancer ID vers la gauche en s'arrêtant sur tous les nombres plus petits que le pivot.
- Quand ID et IG sont arrêté tous le deux, on échange leurs valeurs, puis ils reprennent leur progression.
- Quand ID et IG se croisent "la partie est finie". Il faut placer le pivot à sa place entre les deux parties.

- Sur l'exemple : IG s'arrete en 3 sur le 10, ID s'arrête en 8 sur le 4. On échange les valeurs (en gras) et on continue

5	3	**10**	6	15	7	16	**4**	8
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

avant changement

5	**3**	**4**	6	15	7	16	**10**	8
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

après changement

- On reprend la progression, IG s'arréte sur la valeur 15 d'indice 5, ID s'arrête sur la valeur 7 d'indice 6, on échange les deux valeurs.

5	3	4	6	**15**	**7**	16	10	8
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

avant changement

5	3	4	6	**7**	**15**	16	10	8
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

après changement

- On reprend la progression, tout de suite ID et IG se croisent, ID est en 5 et IG en 6. C'est fini. Il n'y a plus qu'à placer le pivot au bon endroit. On echange IG et IP.

5	3	4	6	7	**15**	16	10	**8**
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

avant changement

5	3	4	6	7	**8**	16	10	**15**
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

après changement

QuickSort

```
public class QuickSort
{  private Comparator ordre;
   private Object[] tab;
   public QuickSort(Object[] tab, Comparator ordre) {}
   public static void sort(Object[] tab, Comparator ordre) {}
   private void quicksort(int deb, int fin){}
   private int partition(int deb, int fin){}
   private boolean estTrie() {}
}
```

1. Programmer le constructeur.
2. Programmer `partition(int bas, int haut)` qui partitionne le tableau et renvoie la position du pivot.
3. Écrire la méthode `estTrie()` qui renvoie `true` si le tableau est bien trié.
4. Programmer `quicksort(int deb, int fin)` qui (lorsqu'il n'est pas déjà trié) trie de manière récursive le sous tableau de tab délimité par les indices `deb` et `fin`.
5. Écrire la méthode `sort(Object[] tab, Comparator ordre)` qui crée une instance de `QuickSort`, puis trie le tableau tab suivant l'ordre fourni par le comparateur `ordre`.

8 Tri par fusion

Écrire un programme qui ordonne un tableau d'objets en utilisant l'interface `Comparator` et la méthode de tri par fusion. Le principe est le suivant : on coupe le tableau en deux parties de même taille à une case près, puis on trie chacune des deux parties. On fusionne ensuite les deux tableaux triés.

Fusion

```
public class Fusion
{
 private Comparator ordre;
 private Object[] tab;
 public Fusion(Object[] tab, Comparator ordre) {}
 public static void sort(Object[] tab, Comparator ordre) {}
 private void triFusion(int deb, int fin) {}
 private void fusion(int deb1, int fin1, int fin2) {}
}
```

1. Programmer le constructeur.
2. Programmer `fusion(int deb1, int fin1, int fin2)` qui fusionne les deux sous- tableaux délimités par deb1 et fin1 d'une part, et par fin1+1 et fin2 d'autre part. Pour cela, on recopiera le premier sous tableau dans un tableau temporaire, et écrira directement dans le tableau initial le résultat de la fusion.
3. Programmer `triFusion(int deb, int fin)` qui trie de manière récursive le sous tableau de tab délimité par les indices deb et fin.
4. Écrire la méthode `sort(Object[] tab, Comparator ordre)` qui crée une instance de `Fusion`, puis trie le tableau `tab` suivant l'ordre ordre.

9 le Tri par Comptage

Ce tri est le tri le plus rapide, puisqu c'est un tri en N. C'est un descendant simplifié du tri "Radix" inventé Par H. Hollerith avec les tabulatices. Ce tri n'est utilisable que lorsque les clefs de tri ont une valeur bornée (quelque millers). Ce tri admet sans pbs des valeurs multiples de la clef.

9.1 Les principes

Ils consistent à céer une table auxiliaire dont la taille est égale au nombre de clefs possible, et dans chaque case de cette table on indique le nb de fois ou une clef apparait dans le tableau à trier. Ces valeurs sont ensuites sommeés, les valeurs sommées vont servir de point de départ pour recopier le tableau inirial dans le tableau triè.

9.2 Exemples

: soit la suite d'entier à trier `5,3,6,4,2,4,5,1,3,4,2,1` la construction des tableaux `Nb` et `Nbs` va donner

3	3
2	5
3	8
2	11

10 Des cartes

On représente une carte à jouer par la classe suivante :

Cartes

```
public class Carte {
private int couleur;
private int valeur;

public static final int PIQUE = 0;
public static final int COEUR = 1;
public static final int CARREAU = 2;
public static final int TREFLE = 3;
public static final int VALET = 11;
public static final int DAME = 12;
public static final int ROI = 13;

public Carte(int genre, int valeur) {
          this.couleur = couleur;
          this.valeur = valeur;
}
public String toString() {
       String[] valeurs = {"Valet", "Dame", "Roi"};
       String[] couleurs = {"Pique", "Coeur", "Carreau", "Trêfle"};
       String nom;
```

```
        if(valeur >= VALET)
                        nom = valeurs[valeur -VALET];
            else
                        nom = "" + valeur;
             return nom + " de " + couleurs[couleur];
         }
}
```

Écrire une classe `ComparatorCarte` implémentant l'interface `Comparator` qui permet de comparer deux cartes. On supposera que les cartes les plus fortes sont celles de couleur Pique, puis les Coeur, puis les Carreaux et enfin les Trèfles. Pour une couleur donnée, la carte la plus faible est l'As et la carte la plus forte est le Roi.

11 Utilisation des tris

Écrire une classe `TesteTriComparator` afin de tester les tris programmés précédemment. La méthode `jeu52cartes()` renvoie un tableau de 52 cartes mélangées. La méthode `randInteger(int taille)` renvoie un tableau d'objets `Integer` générés aléatoirement de dimension taille.

La méthode `tabVersChaine(Object[] tab)` renvoie une chaine de caractères représentant le tableau passé en argument. La méthode `main(String[] args)` crée un tableau de Carte et un tableau d'`Integer`, les affiche, les trie, puis affiche le résultat.

TesteTriComparator

```
import java.util.*;
public class TesteTriComparator {
        public static Carte[] jeu52cartes() {}
        public static Integer[] randIntegers(int taille) {}
        public static String tabVersChaine(Object[] tab) {}
        public static void main(String[] args) {}
}
```

Programmation java objet
DEMIEE 2ème Année 2013-2014

Feuille 5 - Graphisme

1 Analyse de programmes

Donner le dessin produit par le programme suivant : (on rappelle que les composantes d'un objet Color sont Rouge, Vert et Bleu)

ProgrammeGraphique

```
1
2
3        import javax.swing.*;
4        import java.awt.*;
5        public class ProgrammeGraphique extends JPanel {
6           public void paintComponent(Graphics g) {
7              super.paintComponent(g);
8              g.setColor(new Color(255,0,0));
9              g.drawRect(10,10,150,175);
10
11             g.setColor(new Color(0,255,0));
12             g.fillOval(60,20,20,20);
13
14             g.setColor(new Color(0,0,255));
15             g.fillRect(60,40,20,60);
16             g.fillRect(50,100,40,10);
17
18             g.setColor(new Color(255,0,255));
19             g.fillRect(50,110,10,40);
20             g.fillRect(80,110,10,40);
21             g.fillRect(30,50,30,5);
22             g.fillRect(80,50,30,5);
23          }
24          public static void main(String[] args) {
25             JFrame fenetre=new JFrame();
26             fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27             JPanel dessin=new ProgrammeGraphique();
28             dessin.setPreferredSize(new Dimension(200,250));
29             fenetre.getContentPane().add(dessin,BorderLayout.CENTER);
30             fenetre.pack();
31             fenetre.setVisible(true);
32          }
33       }
34
```

2 Programmation

2.1 Damier

1. Ecrire un programme complet (la classe et une méthode principale) qui permet de dessiner un damier de taille quelconque (n x n) et dont la taille du carré élémentaire est égale à 10.
2. Modifiez le programme précédent afin que le damier soit toujours affiché dans la fenêtre de dessin même si l'utilisateur change dynamiquement (à la souris par exemple) la taille de la fenêtre.

2.2 Changement de Repère

Le repère utilisé par Java (à gauche) n'est pas un repère usuel, en particulier à cause de son orientation. Ecrire un objet graphique qui permet de convertir des coordonnées depuis le repère mathématique (à droite) vers le repère de l'écran. Par exemple, on tracera le segment rejoignant les points (-1 ;0) et (2 ;1).

Segment

```
import java.awt.*;
import javax.swing.*;
public class DessinSegment
      extends JPanel {
         private double xmin=-3;
         private double xmax=3;
         private double ymin=-2;
         private double ymax=1;
         public int xMathToGraph(double x) {
               // à compléter
         }
         public int yMathToGraph(double y) {
               // à compléter
         }
         public void paintComponent(Graphics g) {
               super.paintComponent(g);
               // à compléter
         }
    }
```

3 Sujet de TP : Création d'une interface graphique

Le but de cet exercice est d'arriver à construire une application de dessin. On vous donne les classes suivantes : la classe Point et la classe Truc02 qui permet d'afficher un cadre vide.

Point

```
    public class Point {
         public static final int MAX_X = 400;
         public static final int MAX_Y = 400;
```

```
5          private int x, y;
6         public Point(int a, int b) {
7              x = a;
8              y = b;
9         }
10        public Point() {
11             x = (int)(Math.random() * MAX_X);
12             y = (int)(Math.random() * MAX_Y);
13        }
14        public int getX() {
15             return x;
16        }
17        public int getY() {
18             return y;
19        }
20        public String toString() {
21             return "(" + x + "," + y + ")";
22        }
23    }
```

Truc02

```
1  import javax.swing.*;
2
3  public class Truc02 extends JFrame {
4
5      public Truc02() {
6          super("Version 2");
7          setSize(Point.MAX_X, Point.MAX_Y);
8          setVisible(true);
9      }
10
11     public static void main(String[] args) {
12         JFrame cadre = new Truc02();
13     }
14 }
```

1. Ajouter au cadre un panneau (JPanel) supérieur rose, portant deux boutons (JButton) (au hasard et effacer) et un panneau central.
2. Ajouter aux deux boutons un "ActionListener" qui permet d'afficher sur la console le nom du bouton qui a été préssé, de deux manières différentes :
   (a) En utilisant un même écouteur.
   (b) En créant un ecouteur spécifique à chaque bouton
3. Activer le bouton de fermeture de la fenêtre.
   (a) En détectant des actions sur la case de fermeture
   (b) En utilisant les adaptateurs (WindowAdapter)

   (c) la méthode setDefaultCloseOperation (si la fermeture du cadre doit nécessairement se traduire par la terminaison du programme)

4. En utilisant la méthode paint, ajouter au bouton au hasard, un "ActionListener" qui permet de dessiner une ligne. Chaque nouvelle ligne dessinée aura comme l'une des extrémités, une extrémité de la ligne précèdente. Vous stockerez tous les points dans une structure de données.
5. Ajouter au bouton effacer, un "ActionListener" qui permet d'effacer la dernière ligne dessinée.
6. En utilisant un auditeur d'événements souris (MouseListener), dessiner une nouvelle ligne en cliquant sur un point de l'interface graphique.

4 En TP

1. Implémenter les différentes interfaces.
2. Programmer une interface pour le Tic Tac Toe.