Programmation Objet

Table des matières

1 Structures de contrôle

Les programmes suivants doivent être exécutés directement depuis la console.

1. Écrire une fonction `String trans(int n)` qui prend en paramètre un entier et le transforme en chaîne de caractères en utilisant le modulo (%) de la façon suivante : un multiple de 5 est remplacé par fizz, un multiple de 7 par buzz et un multiple des deux par fizzbuzz. Écrire un programme qui affiche ce résultat pour tous les entiers compris entre 1 et 100 (inclus).
2. Écrire une méthode `main` qui affiche dans l'ordre inverse les arguments de la ligne de commande. Le résultat doit être le suivant :

   ```
   java Reverse ab cd def ghk
   ghk def cd ab
   ```

   Modifier ce programme pour que, de plus, chaque argument soit lui-même affiché à l'envers :

   ```
   java Reverse2 ab cd def ghk
   khg fed dc ba
   ```

3. Écrire un programme qui trie un tableau d'entiers en utilisant la méthode dite de sélection : une première fonction `int posMin(int t[], int d, int f)` retourne la **position** du plus petit élément sur la zone `[d,f]`, 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.
   Écrire une autre version de ce programme pour trier un tableau de chaînes de caractères.

2 Objets simples

2.1 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

```
public class PointR2 {
  public double x;
  public double y;
  public PointR2(double a, double b)
  public PointR2()
  public double distance(PointR2 p)
  public boolean equals(Object o)
  public String toString()
}
```

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

1. Programmer les constructeurs et les méthodes en accord avec la documentation suivante. Programmer simultanément une méthode `main` pour tester les méthodes au fur et à mesure.
   (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 définit maintenant un rectangle par deux points aux extrémités d'une diagonale, mais **sans aucune hypothèse sur la disposition de ces deux points**. Le squelette de la classe `Rectangle` est le suivant :

Rectangle

```
public class Rectangle {
  public PointR2 ext1;
  public PointR2 ext2;
  public Rectangle()
  public Rectangle(PointR2 p, PointR2 q)
  public Rectangle(PointR2 cig, double lo, double h)
  public double longueur()
  public double hauteur()
  public double perimetre()
  public double surface()
  public Rectangle symetrique()
}
```

   Programmer les constructeurs et les méthodes en accord avec la documentation suivante (avec toujours une méthode `main` pour tester au fur et à mesure) :
   (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 lo, double h)` construit un rectangle dont `cig` est le coin inférieur gauche, `lo` 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) `Rectangle symetrique()` renvoie le rectangle obtenu à partir du rectangle appelant par symétrie par rapport à la première diagonale,

2.2 Statistiques médicales (1)

Dans cet exercice, on se propose d'écrire une classe utile à la manipulation de statistiques médicales. Un test est composé de trois éléments : le nom du patient concerné, la date de prélèvement, représentée par un entier à 8 chiffres de la forme `aaaammjj` avec l'année sur 4 chiffres, puis le mois et le jour sur deux chiffres chacun, et un taux (d'anticoagulant dans le sang) pour un patient en cours de traitement, qui est donc un nombre compris entre 0 et 1.

La classe Test est donnée par son squelette, où les variables d'instance `nom`, `date` et `taux` donnent respectivement le nom du patient, la date du prélèvement et le taux obtenu. La variable `tauxAlerte` représente le seuil en dessous duquel le taux ne devrait pas descendre.

Test

```
public class Test {
    private static double tauxAlerte=0.22;
    private String nom;
    private int date;
    private double taux;

    public Test(String n, int d, double t)
    public Test()
    public String getNom()
    public double getTaux()
    public int getDate()
    public String toString ()
    public boolean isTest()
    public boolean alerte()
    public boolean equals(Object o)
    public double compareTaux(Test t)
}
```

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

1. Le constructeur à 3 paramètres construit un test dont le nom associé est la chaîne `n`, la date est `d` et le taux est `ta`. Le constructeur `Test()` construit le test ayant la chaîne vide (`""`) comme nom, 0 comme date et -1 comme taux.
2. Les méthodes "observateurs" `getNom()`, `getDate()` et `getTaux()` retournent respectivement le nom, la date et le taux du test appelant.
3. La méthode `toString()` renvoie une chaîne de la forme `[nom, date : taux]`.
4. La méthode `isTest()` vérifie que le taux est bien compris entre 0 et 1. Elle renvoie `true` dans ce cas et `false` sinon.
5. La méthode `alerte()` renvoie `true` si le taux du test appelant est inférieur ou égal au seuil d'alerte (tout en restant un taux), `false` sinon.
6. La méthode `compareTaux(Test t)` renvoie la différence de taux entre le test appelant et le test en paramètre, de sotre que l'appel `t1.compareTaux(t2)` renvoie un double strictement positif (respectivement négatif) si le taux du test appelant est strictement supérieur (respectivement inférieur) à celui du test en paramètre et 0 si les taux sont égaux.

7. La méthode `equals(Object o)` teste l'égalité de deux tests. Pour cela, on suppose qu'il n'y a qu'un prélèvement par jour et par personne, le test d'égalité porte donc uniquement sur le nom et la date.

2.3 Exécution de tâches (1)

Une tâche est caractérisée par trois éléments : son numéro d'identification, sa durée et le temps restant à exécuter. Le temps restant est initialement égal à la durée et il décroit à chaque fois que la tâche est exécutée pendant un certain temps. La tâche est terminée lorsque son temps restant est égal à 0. La classe correspondante `Task` est définie par :

```
public class Task {
    private int id;
    private int duree;
    private int reste;
    public Task(int n, int d){}
    public int getId(){}
    public boolean fini(){}
    public void exec(int t){}
    public boolean equals(Object o){}
    public String toString(){}
    public static void main(String[] arg){
        Task ta = new Task(1,2);
        ta.exec(1);
        System.out.println(ta);
    }
}
```

Programmer cette classe avec la documentation suivante :

(a) Le constructeur `Task(int n, int d)` construit une tâche ayant pour numéro `n` et pour durée `d`. Il initialise aussi le temps restant.

(b) La méthode "observateur" `int getId()` retourne le numéro de la tâche. La méthode `boolean fini()` retourne `true` si et seulement si la tâche est terminée.

(c) La méthode `void exec(int t)` simule une exécution de la tâche appelante pour une durée de `t`. Elle met donc à jour le temps restant. Celui-ci est mis à 0 si la durée `t` est supérieure au temps qui restait avant l'exécution.

(d) La méthode `boolean equals(Object o)` teste l'égalité de deux tâches : deux tâches sont égales si et seulement si elles ont même numéro.

(e) La méthode `String toString()` retourne une chaîne de caractères décrivant une tâche sous la forme `[n;d;r]`, où `n` est le numéro, `d` la durée et `r` le temps restant.

Donner le résultat d'exécution du programme.

3 Objets et types élémentaires en paramètres

1. On considère la définition d'objet suivante :

A

```
public class A {
    public int x;
    public A(int x){
        this.x=x;
    }
    public int f(){
        return x+3;
    }
    public A g(A a){
        return new A(2*x+3*a.x);
    }
}
```

ainsi que le programme :

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 }
```

(a) Dessiner l'état de la mémoire juste après l'exécution de la dernière ligne du programme `TestA`.

(b) Donner l'affichage produit par `TestA`.

(c) Quel est l'affichage produit 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) Dessinez l'état de la mémoire juste après l'exécution de la dernière ligne du programme.

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

(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) Dessiner l'état de la mémoire pendant l'exécution de la ligne 31.

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

4 Stockage d'objets par tableaux

4.1 La classe PileT

Programmer une classe `PileT`, contenant des objets queconques. Les variables d'instance sont : un tableau et l'indice du sommet de pile, qui représente aussi le nombre courant d'éléments dans la pile. Les méthodes, outre le constructeur qui fournit une pile vide, correspondent aux opérations habituellement associées aux piles : tester si la pile est vide (`isEmpty()`), empiler un élément (`push(Object o)`), dépiler (`pop()`) qui retourne le sommet de pile et le retire, et lire le sommet de la pile (`top()`). Ajouter également une méthode `toString()`.

4.2 Statistiques médicales (2)

Sur le modèle de la pile, programmer une classe `EchantillonT`, contenant des objets de la classe `Test`. Les variables d'instance sont : un tableau et le nombre courant d'éléments présents (représentant également l'indice du premier emplacement libre). Les méthodes, outre le constructeur qui construit une liste vide, correspondent aux opérations utiles pour manipuler un tel échantillon (modifications, affichage des taux dangereux, etc.). Ajouter également une méthode `toString()`.

5 Récursivité statique

1. Programmer le tri par sélection récursivement. Pour cela, la procédure de tri prendra le début de la zone en paramètre, comme la fonction `posMin`.
2. Écrire une version itérative et une version récursive d'une fonction qui retourne la somme des éléments d'un tableau d'entiers.
3. Donner les affichages produits par l'exécution du programme suivant :

Compte

```
public class Compte {
    public static void compte1(int n){
        if(n>0) {
            compte1(n-1);
            System.out.println(n);
        }
    }
    public static void compte2(int n){
        if(n>0) {
            System.out.println(n);
            compte2(n-1);
        }
    }
    public static void main(String[] args) {
        compte1(5);
        compte2(5);
    }
}
```

4. Écrire un programme donnant les opérations successives à réaliser pour résoudre le problème des tours de Hanoï avec n disques superposés.

6 Objets récursifs

6.1 La classe PileM

A l'aide de la classe `Maillon` définie en cours, programmer une classe `PileM` réalisant les méthodes de pile par chaînage.

6.2 La classe Abr

On rappelle qu'un arbre binaire sur un ensemble A est défini de façon inductive par :
— $\emptyset$ est un arbre binaire
— si $a \in A$ et si g et d sont deux arbres binaires alors $t = (a, g, d)$ est un arbre binaire.
Dans ce cas, g est appelé le sous-arbre gauche de t et d le sous-arbre droit de t, et a est l'étiquette de la racine de t.
Un arbre binaire de recherche est un arbre binaire t qui possède de plus la propriété suivante : l'ensemble A est totalement ordonné (on prendra pour A l'ensemble des chaînes de caractères dans l'exercice) et pour tout sous-arbre $t' = (b, g', d')$ de t, toutes les étiquettes du sous-arbre gauche g' de t' sont inférieures ou égales à b et toutes les étiquettes du sous-arbre droit d' de t' sont strictement supérieures à b.

Programmer une classe `Abr`, représentant des arbres binaires de recherche, avec le squelette suivant :

Abr

```
1  public class Abr {
2      private String sommet;
3      private Abr gauche;
4      private Abr droit;
5
6      public Abr()
7      public Abr(String c)
8      public Abr(String c, Abr g, Abr d)
9      public boolean isEmpty()
10     public String getSommet()
11     public Abr getG()
12     public Abr getD()
13     public int height()
14     public void add(String c)
15     public void affPP()
16     public String toString()
17
18     public static void main(String[] arg){
19         String[] s={"Victor", "Boris", "Emile", "Zoe", "Ursule", "Toto"};
20         Abr b= new Abr();
21         for (int i=0; i<s.length; i++)
22             b.ajoute(s[i]);
23         b.affPP();
24         System.out.println();
25     }
26 }
```

et la documentation :

1. Les constructeurs retournent respectivement un arbre vide, un arbre réduit à sa racine donnée en paramètre et un arbre dont les trois éléments sont donnés en paramètre.
2. la méthode `isEmpty()` retourne vrai si et seulement si l'arbre est vide, un arbre étant considéré comme vide si son sommet est vide.
3. La méthode `height()` renvoie la hauteur de l'arbre.
4. La méthode `add(String c)` insère un élément de façon à conserver la propriété d'arbre binaire de recherche.
5. La méthode `affPP()` affiche les étiquettes de l'arbre en ordre croissant en utilisant un parcours en profondeur.
6. la méthode `toString()` renvoie une chaîne représentant l'arbre "à la Lisp", c'est-à-dire de la forme (récursive) : (racine filsgauche filsdroit).

6.3 Exécution de tâches (2) : listes circulaires

On considère la classe `Maillon` suivante, dans laquelle un maillon est constitué d'un entier et d'une référence vers un maillon :

Maillon

```
1  public class Maillon {
2      private int num;
3      private Maillon suivant;
4      public Maillon(int i, Maillon s){
5          num=i;
6          suivant=s;
7      }
8      public String toString(){
9          return String.valueOf(num);
10     }
11     public int getCont(){
12         return num;
13     }
14     public Maillon getNext(){
15         return suivant;
16     }
17     public void setNext(Maillon m){
18         suivant=m;
19     }
20    public static void main(String[] arg){
21        Maillon m1= new Maillon(3,null);
22        Maillon m2 = new Maillon(5,m1);
23        System.out.println(m2);
24        System.out.println(m2.suivant);
25        m1.setNext(m2);
26        System.out.println(m1.suivant);
27    }
28 }
```

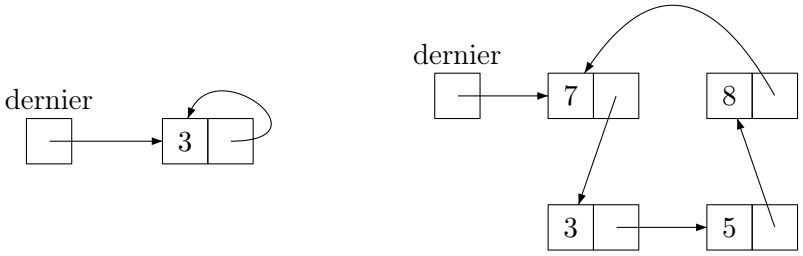
(a) Dessiner l'état de la mémoire à la fin du programme.
(b) Donner le résultat d'exécution du programme.

(c) Donner une représentation schématique des relations entre les entiers impliqués.

On choisit maintenant de ranger des entiers dans des listes circulaires. On définit pour cela la classe `ListeC`, qui utilise la classe `Maillon`, de la façon suivante :

```
public class ListeC {
    private Maillon dernier;
    public ListeC(){}
    public boolean isEmpty(){}
    public Maillon premier(){}
    public int numPremier(){}
    public void add(int n){}
    public void remove(){}
    public void lNext(){}
    public String toString(){}
    public static void main(String[] arg){
        ListeC lc = new ListeC();
        System.out.println(lc);
        lc.add(3);
        lc.add(5);
        lc.add(8);
        System.out.println(lc);
        lc.lNext();
        System.out.println(lc);
        lc.remove();
        System.out.println(lc);
    }
}
```

Une liste est repérée par son **dernier** élément, elle est vide lorsque `dernier` vaut `null`. Le premier élément est donc celui qui suit le dernier. Une liste circulaire contenant un seul élément est représentée ci-dessous à gauche, tandis qu'une liste circulaire contenant la suite d'entiers $3; 5; 8; 7$ est représentée ci-dessous à droite. Sur cet exemple, l'élément qui contient l'entier 7 est le dernier, tandis que l'élément qui contient l'entier 3 est le premier.



Programmer cette classe en respectant la documentation suivante.

(a) Le constructeur `ListeC()` construit une liste vide et la méthode `boolean isEmpty()` renvoie `true` si et seulement si la liste est vide.

(b) La méthode `Maillon premier()` retourne une référence vers le premier maillon, c'est-à-dire celui qui suit immédiatement le dernier, lorsque la liste n'est pas vide, `null` sinon. La méthode `int numPremier()` retourne l'entier contenu dans le premier maillon, -1 si la liste est vide. Pour la liste ci-dessus, la valeur retournée serait donc 3.

(c) La méthode `void add(int n)` insère dans la liste un nouveau maillon contenant l'entier `n`. Ce nouveau maillon est placé juste après celui qui est pointé par `dernier`, et

devient ensuite le dernier. Sur l'exemple ci-dessus, insérer un maillon contenant 2 donnerait la liste $3; 5; 8; 7; 2$ de la figure 1.

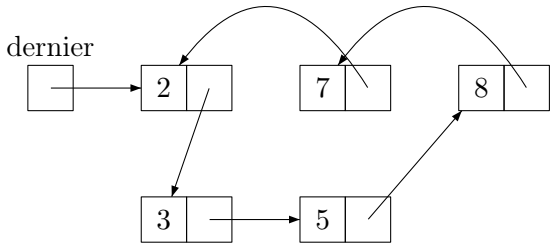


FIGURE 1 – Insérer un nouvel élément

(d) La méthode `void remove()` supprime le **premier** élément de la liste. À partir de la liste précédente, appliquer cette méthode produirait la liste $5; 8; 7; 2$ (figure 2).

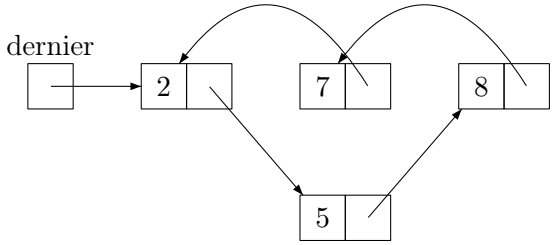


FIGURE 2 – Supprimer un élément

(e) La méthode `void lNext()` décale le dernier élément de la liste d'un maillon vers l'avant. Sur la liste qui précède, on obtiendrait $8; 7; 2; 5$ (figure 3).

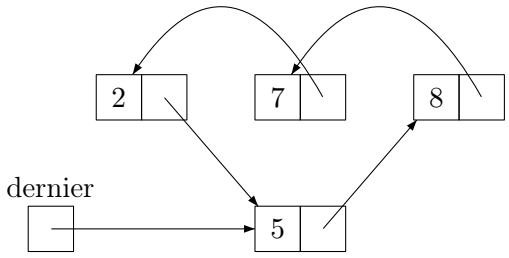


FIGURE 3 – Décalage vers l'avant

(f) La méthode `String toString()` retourne une chaîne de caractères contenant les entiers de la liste dans l'ordre, en commençant par le **premier**.

Donner le résultat d'exécution du programme.

7 Classes enveloppes, listes et surcharge d'opérateurs

7.1 Grands entiers

On souhaite pouvoir manipuler des entiers naturels ayant un nombre quelconque de chiffres. Ceci permet par exemple de calculer des factorielles sans être limité par la taille des entiers standards. Pour cela, on définit une classe `GrandEntier`, où un entier est représenté par une liste d'entiers contenant la suite de ses chiffres (en base 10), sous forme d'`Integer`. 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 à l'indice 1, etc. (c'est-à-dire dans l'ordre inverse de celui qui sera utilisé pour l'affichage). 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 4492105337 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.

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

GrandEntier

```
import java.util.*;
public class GrandEntier {
    private List<Integer> liste;

    public GrandEntier(int n){}
    public GrandEntier(){}
    public GrandEntier(String s){}
    public GrandEntier(GrandEntier g){}
    public int nbChiffres(){}
    public int chiffre(int i){}
    public String toString(){}
    public boolean equals(Object o){}
    public GrandEntier somme(GrandEntier g){}
    public GrandEntier mult(int k){}
    public GrandEntier mult10(int p){}
    public GrandEntier mult(GrandEntier g){}
    public GrandEntier puis(int k){}
    public static GrandEntier fact(int k){}

    public static void main(String[] args){
        GrandEntier g=new GrandEntier();
        System.out.println(g);
        System.out.println(g.nbChiffres());
        GrandEntier h=new GrandEntier(0);
        System.out.println(h);
        System.out.println(h.nbChiffres());
        System.out.println(h.equals(g));
        String s="3452";
        GrandEntier m=new GrandEntier(s);
        GrandEntier mcopie= new GrandEntier(m);
```

```
        System.out.println(m);
        System.out.println(mcopie);
        h=m.somme(g);
        System.out.println(g);
        System.out.println(m);
        g=m.mult(4);
        System.out.println(g);
        System.out.println(m.mult10(1));
        g=new GrandEntier(23);
        System.out.println(m.mult(g));
        h=g.puis(2);
        System.out.println(h);
        System.out.println(Math.pow(23,2));
        for(int i=0;i<15; i++)
            System.out.println(fact(i));
    }
}
```

Programmer les constructeurs et les méthodes de la classe `GrandEntier` en respectant la documentation suivante. Testez vos méthodes avec le programme donné, en le complétant au fur et à mesure.

1. Le constructeur sans paramètre consruit l'entier 0, qui correspond à une liste vide.
   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.
   Le constructeur `GrandEntier(String s)` construit l'entier correspondant à la chaîne `s`.
   Le constructeur `GrandEntier(GrandEntier g)` construit un grand entier en **recopiant** celui donné en paramètre.
2. La méthode `nbChiffres()` renvoie le nombre de chiffres du grand entier appelant.
   La méthode `chiffre(int i)` renvoie le chiffre en position `i` (toujours en commençant par les unités).
3. La méthode `toString()` retourne une chaîne de caractères correspondant au grand entier représenté. Attention, cette chaîne doit être conforme à l'écriture habituelle.
4. La méthode `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.
5. Les opérateurs usuels renvoient les résultats attendus : somme, produit, puissance (avec un exposant entier standard seulement), etc. Ces opérations sont réalisées comme on le fait avec un papier et un crayon. La multiplication utilise deux méthodes auxiliaires : multiplication par un chiffre et multiplication par une puissance de 10 (par décalage).
6. La méthode de classe `fact(int k)` retourne un grand entier qui représente `k`!, où `k` est un entier usuel.

7.2 Exécution de tâches (suite et fin)

On veut maintenant ranger les tâches considérées dans la section 2 dans une liste, qui sera réalisée par une `ArrayList`. La classe `ListeT` est définie par :

```
import java.util.*;
public class ListeT {
    private List<Task>  listeTask;
    public ListeT(){}
    public int nbTask(){}
    public int indexOf(int num){}
    public Task getTask(int pos){}
    public void add(Task ta){}
    public void chTask(int pos, Task ta){}
    public void remove(int pos){}
    public String toString(){}


    public static void main(String[] arg){
        ListeT lT= new ListeT();
        lT.add(new Task(0,3));
        Task ta = new Task(1,2);
        lT.add(ta);
        lT.add(new Task(2,6));
        System.out.println(lT);
        Task tb = new Task(2,7);
        int place=lT.indexOf(2);
        lT.chTask(place,tb);
        System.out.println(lT.getTask(0));
        System.out.println(lT);
    }
}
```

1. Programmer cette classe en respectant la documentation suivante, et en utilisant les méthodes de la classe `ArrayList` autant que possible :

   1. Le constructeur `ListeT()` construit une liste vide et la méthode `int nbTask()` retourne le nombre de tâches de la liste.
   2. La méthode `int indexOf(int num)` retourne l'indice de la tâche de numéro `num` dans la liste (`-1` si elle n'est pas présente) et la méthode `Task getTask(int pos)` retourne la tâche d'indice `pos` dans la liste.
   3. La méthode `void add(int Task ta)` ajoute la tâche `ta` dans la liste, la méthode `void chTask(int pos, Task ta)` modifie la tâche d'indice `pos` dans la liste en la remplaçant par `ta` et la méthode `void remove(int pos)` retire la tâche d'indice `pos` de la liste.
   4. La méthode `String toString()` retourne une chaîne de caractères décrivant la liste.

2. Donner le résultat d'exécution du programme.
3. On veut finalement simuler une exécution pour un ensemble de tâches. On fixe une durée d'exécution, appelée le *quantum*, qui sera allouée à chaque tâche à son tour. La première tâche de la liste circulaire est d'abord exécutée, pour une durée égale au quantum. Si elle est terminée, elle est retirée du système. Puis, on passe à la tâche suivante, de manière circulaire

(d'où le nom de tourniquet).
Par exemple, supposons qu'on dispose de trois tâches `[0,5,5]`, `[1,2,2]`, `[2,3,3]`, avec un quantum de 1. La simulation exécute d'abord la tâche `0` pour 1 unité de temps, elle devient donc `[0,5,4]`. On obtient ensuite `[1,2,1]` pour la tâche `1`, puis `[2,3,2]` pour la tâche `2`, puis `[0,5,3]` pour la tâche `0` et `[1,2,0]` pour la tâche `1`. Celle ci étant terminée, on la retire. En poursuivant les exécutions successives, on obtient maintenant `[2,3,1]`, `[0,5,2]`, puis `[2,3,0]`. La tâche `2` est donc retirée à son tour et, finalement, il ne reste plus que la tâche `0`, qui s'exécute deux fois de suite.

On considère pour cela un système qui comporte une liste de tâches `listetaches`, une liste circulaire d'entiers `tourniquet`, dans laquelle seront rangés les numéros d'identification des tâches à exécuter, et un entier `quantum`. La classe `Sys` est définie par :

```
public class Sys {
    private ListeT listetaches;
    private ListeC tourniquet;
    private int quantum;


    public Sys(){
        quantum=1;
        listetaches = new ListeT();
        tourniquet = new ListeC();
        for(int i=0;i<5;i++){
            listetaches.add(new Task(i,(int)(Math.random()*10)));
            tourniquet.add(i);
        }
    }
    public boolean finexec(){}
    public String toString(){}
    public void execQuantum(){}
    public static void main(String[] arg){
        Sys s=new Sys();
        System.out.println(s);
        while(!s.finexec()){
            s.execQuantum();
            System.out.println(s);
        }
    }
}
```

La simulation de l'exécution des tâches se fait par le programme principal, où chaque étape (méthode `void execQuantum()`) consiste en l'exécution d'une tâche de la liste circulaire, pour une durée égale au quantum.
Programmer les méthodes suivantes de la classe `Sys`.

(a) La méthode `boolean finexec()` retourne `true` si et seulement si l'exécution est terminée (tourniquet vide).

(b) La méthode `String toString()` retourne une chaîne de caractères décrivant le système.

(c) La méthode `void execQuantum()` effectue une étape de la simulation : la **première** tâche (et non la dernière) de la liste circulaire est exécutée pour une durée égale au

`quantum`. Une mise à jour du système est ensuite effectuée : si la tâche considérée est terminée, elle est retirée de la table et du tourniquet, sinon la table est mise à jour et on passe à la tâche suivante du tourniquet.

8 Interfaces Comparable et Comparator, héritage

8.1 Nombres rationnels

On souhaite pouvoir manipuler des nombres rationnels et au besoin trier des suites de rationnels. Pour cela, on définit une classe `Rat` qui réalise l'interface `Comparable` et dont les variables d'instance sont deux entiers `num` et `den` représentant respectivement le numérateur et le dénominateur de la fraction considérée.

1. Programmer cette classe avec les méthodes suivantes :

```
public static int pgcd(int p, int q)
public Rat()
public Rat(int n)
public Rat(int p, int q)
public String toString()
public Rat plus(Rat r)
public Rat fois(Rat r)
public double doubleVal()
```

   ainsi que la méthode nécessaire pour réaliser l'interface considérée, en respectant la documentation ci-dessous.

   (a) Le constructeur sans paramètre construit l'entier 0, Le constructeur `Rat(int n)` construit l'entier `n` donné en paramètres et le constructeur `Rat(int p, int q)` construit le rationnel p/q sous forme d'une fraction irréductible (en utilisant la méthode pgcd) dont le dénominateur est positif.

   (b) La méthode `toString()` retourne une chaîne de caractères de la forme `p/q` en tenant compte des cas particuliers.

   (c) Les méthodes `plus`, `fois` renvoient respectivement la somme et le produit du rationnel appelant et du rationnel en paramètre et la méthode `doubleVal()` renvoie la valeur du rationnel calculée comme un `double`.

2. Écrire une méthode `main` qui calculera en particulier la somme des 10 premiers termes de la suite $1/2^n$ et qui triera un tableau de rationnels. Utiliser pour cela des méthodes statiques de la classe `Arrays` : `sort(T[] tab)` qui trie un tableau d'objets de type `T` à condition que la classe réalise l'interface `Comparable`, et `toString()`, qui renvoie un tableau sous forme de chaîne de caractères.

8.2 Figures

On considère dans cet exercice une interface `Figure` qui étend l'interface `Comparable`, de façon à pouvoir comparer différentes sortes de figures géométriques selon leur périmètre.

```
interface Figure extends Comparable<Figure> {
  public double perimetre();
```

```
  public double aire();
}
```

1. En réutilisant la classe `PointR2`,
   — reprendre la classe `Rectangle` et l'adapter pour qu'elle réalise l'interface `Figure`,
   — définir une nouvelle classe `Cercle` réalisant également cette interface,
   de sorte que deux figures différentes puissent être comparées par leur périmètre.
2. Écrire un programme de test extérieur à ces deux classes pour trier un tableau contenant différentes sortes de figures (en utilisant les méthodes déjà mentionnées dans l'exercice précédent).

8.3 Statistiques médicales (3)

Dans cet exercice, on voudrait pouvoir trier un échantillon contenant des tests, soit par ordre alphabétique des personnes testées, soit selon les taux pour vérifier les alertes.

1. Programmer une classe `OrdreTest` qui réalise l'interface `Comparator` avec une variable d'instance `critere` et qui compare deux tests soit pour l'ordre alphabétique des noms (si `critere` vaut 0) soit pour l'ordre sur les taux (si `critere` vaut 1).

```
public class OrdreTest implements Comparator<Test> {
    private int critere;

    public OrdreTest(int c){
       critere = c;
    }

    public int compare(Object o1, Object o2){}
}
```

2. Programmer une classe `Echantillon` pour stocker des suites de tests. Cette classe hérite de `ArrayList<Test>`.
3. Ajouter à la classe `Echantillon` une méthode `private void triaux(int crit)` qui trie les échantillons suivant le critère `crit` donné en paramètre.
4. En utilisant des instances adaptées de la classe `OrdreTest`, ajouter également deux méthodes `public void tritaux()` et `public void trinoms()` qui réalisent les tris souhaités.