**Nom :**
**Prénom :**
**Groupe :**

***Algorithmique et programmation***
***mars 2002***

*Les exercices ne sont pas classés par ordre de difficulté et ils sont indépendants les uns des autres.*

---

**Exercice 1** On peut calculer une approximation de $\sqrt{x}$, pour $x > 0$, grâce à la suite récurrente suivante, convergente vers $\sqrt{x}$ :

$$\begin{aligned} y_0 &= 1 \\ y_{n+1} &= 1/2(y_n + x/y_n) \end{aligned}$$

Définir une fonction

```
static double sqrt(double x) { ... }
```

qui calcule une approximation de $\sqrt{x}$ en utilisant cette suite. Quel test d'arrêt choisissez-vous ?

---

**Exercice 2** Pour trier un tableau de $n$ éléments par ordre croissant, on procède de la façon suivante : on sélectionne le plus petit élément et on le place en première position du tableau en le permutant avec l'élément qui s'y trouvait ; on sélectionne ensuite le plus petit des $n - 1$ autres éléments et on le place en seconde position, ainsi de suite. La fonction suivante réalise cet algorithme :

```
static void sélection(Object[] t) {
  for (int i=0; i<t.length; i++) {
    Object premier = t[i];
    Object min = premier;
    int indiceMin = i;
    // recherche de l'élément minimum d'indice >= i
    for (int j=i; j<t.length; j++) {
      if (t[j].compareTo(min) < 0) {
        min = t[j];
        indiceMin = j;
```

```
        }
      }
      // échange des éléments d'indices i et indiceMin
      t[i] = min;
      t[indiceMin] = premier;
    }
  }
```

Ce programme est erroné, à la ligne

```
        if (t[j].compareTo(min) < 0) {
```

Pourquoi ?

Insérez les corrections nécessaires dans le programme ci-dessus pour que cette comparaison ait un sens.

Soit $n$ la longueur du tableau $t$. Quelle est le nombre de comparaisons effectuées, en fonction de $n$, pour l'évaluation de `sélection(t)` ?

Parmi les interfaces suivantes `Collection`, `List`, `SortedSet`, `Set` du paquet `java.util`, quelle est celle qui peut être employée dans cet algorithme de tri, à la place de la structure de tableau ?

Parmi les classes du même paquet réalisant cette interface, quelle est celle qui permet l'implémentattion la plus efficace de cet algorithme ?

Connaissez-vous un algorithme de tri plus efficace que le tri par sélection d'un tableau, quel est son nom et sa complexité ?

---

**Exercice 3** Soit `A` une classe et `B` une classe étendant $A$. Les variables suivantes sont définies :

```
A a = new A( ... );
B b = new B( ... );
```

Dire, pour chacune des affectations suivantes, si elle est correcte ou non, à la compilation, et si elle est correcte ou non, à l'exécution :

| | correcte à la compilation | correcte à l'exécution |
|---|---|---|
| `A a1 = b;` | | |
| `B b1 = a;` | | |
| `B b2 = a1;` | | |
| `A a3 = (A) b;` | | |
| `B b3 = (B) a;` | | |
| `B b4 = (B) a1;` | | |
| `B b5 = (A) a3;` | | |

**Exercice 4** On modélise un compte en banque par une interface qui autorise les opérations suivantes :

- le dépôt d'une somme (de type `double`)
- le retrait d'une somme
- la consultation du solde du compte

Définir une interface `Compte` qui déclare ces opérations.

La classe `CompteCourant` réalise cette interface. Quel champ est nécessaire, doit-il être privé ou public ?

Définir dans cette classe les trois méthodes de l'interface `Compte`, sachant que le retrait n'est possible que si le solde est suffisant. Comment modéliser la situation où le retrait n'est pas possible ?

On considère maintenant le cas d'un compte rémunéré à un taux d'intérêt fixé lors de la création du compte. Chaque dépôt doit maintenant être "estampillé" par sa date. On dispose d'une classe `java.util.Date` telle que l'évaluation de `new Date()` donne la date au moment de l'évaluation. Cette classe

contient une méthode `getTime()` qui donne le nombre de millisecondes écoulées depuis le 1er janvier 1970. Définir une classe `Dépôt`, dont les instances sont caractérisées par un montant et une date, avec un constructeur.

Définir une classe `CompteRémunéré` qui réalise l'interface `Compte`. Quelle structure de données adopter pour conserver l'historique de tous les dépôts ? Quels autres champs sont maintenant nécessaires ? Définir les constructeurs utiles.

Définir la méthode `dépôt` de cette classe.

Les retraits sont maintenant impossibles (à moins de clôturer le compte, ce qu'on ne demande pas de modéliser). Comment représenter cette impossibilité?

On suppose que la rémunération est calculée à chaque opération de consultation du solde (mais pas effectuée : le versement effectif des intérêts ne sera fait qu'à la clôture du compte). Définir la fonction de consultation du solde qui doit retourner la somme des dépôts et des intérêts calculés, au moyen de la formule $me^{rt}$, où $m$ est le montant déposé, $r$ le taux, et $t$ la durée écoulée depuis

le dépôt.

**Exercice 5** Un problème de vacances. Vous souhaitez remplir votre sac à dos avec des objets de type $i$ ($i \in [1, n]$) caractérisés par leur poids $w_i$ et leur utilité $p_i$. Les objets sont en fait sécables et vous pouvez vous contenter de prendre une proportion $x_i$ de l'objet $i$ (le poids est alors $w_i x_i$ et l'utilité $p_i x_i$). Le poids maximum que vous pouvez porter est connu et vous souhaitez trouver les proportions $(x_i)_{i \in [1,n]}$ qui maximisent votre utilité tout en respectant la contrainte de poids.

Quel est le problème mathématique à résoudre ?

Ce problème est modélisé et résolu par la classe `SacADos`, et sa méthode `remplir`, qui détermine la proportion de chaque objet à placer dans le sac à dos :

```
public class SacADos  {
  private List items;
  private static int poidsMax = 20;

  void remplir() { ... }
}
```

Le champ `items` est une liste de `SacADosItem` (Cf. ci-dessous), et le champ `poidsMax` est le poids maximum du sac à dos. On pourra tester cette classe de la façon suivante :

```
  public static void main (String[] args) {
    int[] p = new int[] {25, 24, 15};
    int[] w = new int[] {18, 15, 10};
    SacADos sac = new SacADos(p, w);
    sac.remplir();
    System.out.println(sac);
  }
```

Les objets que l´on va stocker dans le sac à dos seront décrits par la classe `SacADosItem`. Un objet de cette classe est caractérisé par son utilité $p$, son

poids $w$ et la proportion $x$ de cet objet qui sera retenue pour composer le sac à dos. On doit également pouvoir comparer deux `SacADosItem` entre eux en comparant les quantités $p/w$. Écrire la classe `SacADosItem`. Cette classe doit réaliser l'interface `Comparable` et doit comporter des méthodes d'accès pour ses champs privés, notamment

- `getPoids()`, qui renvoie le poids d´un objet.
- `setX(double x)`, qui fixe la proportion retenue pour un objet.

```
class SacADositem                          {



  SacADositem (                ) {



  }





}
```

L´algorithme utilisé pour remplir le sac à dos est le suivant:

```
  void remplir() {
    // On trie la liste des objets en ordre décroissant.
    Collections.sort(this.items, Collections.reverseOrder());
    double reste = this.poidsMax;
    Iterator it = this.items.iterator() ;
    while (it.hasNext()) {
      if (reste > 0) {
        SacADosItem kpi = (SacADosItem) it.next();
        double weight = kpi.getPoids();
        if (weight > reste) {
          kpi.setX(reste/weight);
          reste = 0;
```

```
        } else {
          kpi.setX(1.0);
          reste = reste - weight;
        }
      } else {
        break;
      }
    }
  }
```

Pouvez-vous expliquer ce que fait cet algorithme et à quelle famille générale d´algorithmes il appartient?

Dans la classe `SacADos` pouvez-vous compléter le constructeur et la méthode `toString`:

```
public class SacADos  {
  private List items;
  private static int poidsMax = 20; // poids maximum

  SacADos(int[] p , int[] w) {




  }

  void remplir() { ... }

  public String toString () {




  }
}
```

**Exercice 6** On dispose d'une interface `MoyenDéplacement` et de deux classes la réalisant comme ci-dessous.

```
public interface MoyenDéplacement {
  int coûtUsage(int années);
}

public class Voiture implements MoyenDéplacement {
  // ... champs et constructeurs nécessaires
  public int coûtUsage(int années) {
    // ... définition spécifique  à Voiture
  }
}

public class Train implements MoyenDéplacement {
  // champs et constructeurs nécessaires
  public int coûtUsage(int années) {
    // ... définition spécifique à Train
  }
}
```

On désire maintenant disposer d'une méthode permettant de comparer le coût d'un moyen de déplacement à un autre moyen de déplacement au bout d'un certain nombre d'années d'utilisation. Par exemple, si `maVoiture` est une instance de la classe `Voiture` et `monRER` une instance de la classe `Train`, on pourra comparer lequel est le plus avantageux au bout de 5 ans en évaluant l'expression :

```
maVoiture.estMeilleurMarchéAuFinal(monRER, 5)
```

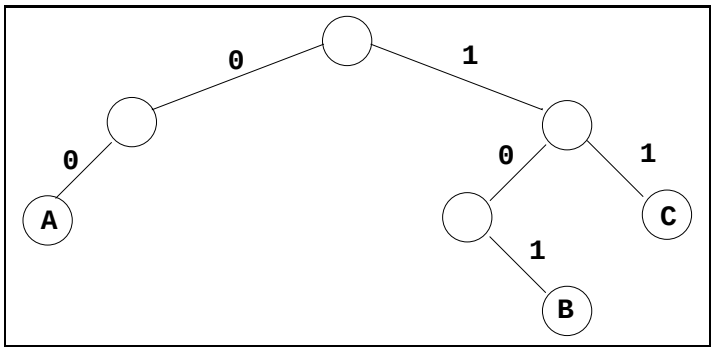
Quelle est la déclaration de cette méthode qui doit être ajoutée à l'interface `MoyenDéplacement` ?

Donnez la définition de la méthode `estMeilleurMarchéAuFinal` dans chacune des classes `Voiture` et `Train`.

Que remarquez-vous ? Peut-on simplifier ce programme en plaçant ces définitions dans une nouvelle classe, et quelle serait sa position dans la hiérarchie des types, par rapport aux trois types déjà définis ?

---

**Exercice 7** Voici un arbre de décodage, correspondant à un code binaire des mots sur l'alphabet $\{A, B, C\}$ :




Décodez le mot binaire `101110011`.

Ce code est-il optimal ? Sinon, dessinez l'arbre de décodage d'un code meilleur que celui-ci, c'est-à-dire qui permette un codage plus compact.