Université de Nice-Sophia Antipolis — POLYTECH
PeiP2 — 2019–2020

Examen de Programmation Objet

Durée : 1h30
Aucun document autorisé
Mobiles interdits **Note importante : le sujet est formé de 3 parties à rédiger sur 3 copies différentes. Sur chacune des copies, vous indiquerez lisiblement votre nom, votre groupe et la partie traitée.**

Le code Java que vous écrirez doit être lisible et commenté avec des affirmations significatives.

Enfin, on rappelle que l'usage des téléphones portabes est strictement interdit.

Partie 1. Classes - Objets

Vous allez modéliser des appartements ayant une certaine superficie, et des immeubles constitués d'appartements.

Appartement

Un appartement a une surface, exprimée en mètres carrés, un occupant (désigné par son nom lorsque le logement est occupé), un booléen à vrai si le logement est actuellement occupé, et quand il l'est, le loyer mensuel qui a été défini entre le propriétaire et le locataire. Pour chaque attribut (ou variable) défini, vous écrirez les méthodes permettant d'accéder en lecture (accesseur/getter) ou en modification (mutateur/setter) à ces attributs. Un objet de type `Appartement`, lorsqu'il est instancié, est toujours défini par le nombre de ses mètres carrés. Mais, au moment de l'instantiation d'un appartement, il n'y a pas forcément d'occupant et donc de loyer mensuel associé. De plus, si l'appartement devient inoccupé, l'occupant et le loyer perçu doivent être effacés.

▶ 1. Écrivez la classe `Appartement` qui possède les propriétés données ci-dessus, ainsi que la méthode `toString` qui affiche les informations suivantes : la superficie et, si le logement est occupé, le nom de l'occupant et le loyer.

```
public class Appartement {
    boolean ocuppé=false;
    String nomloc=null;
    double metrec;
    double loyerm;
```



```
    public Appartement(double m) {metrec=m;}
    public Appartement(double m, String l, double loyerm) {
        this(m);
        this.occupé=true;
        this.loyerm=loyerm;
        this.nomloc=l;
    }
    public String getNomloc() {return this.nomloc;}
    public void setNomloc(String n) {this.nomloc=n;}
    public void setMetrec(double m) {this.metrec=m;}
    public double getMetrec() {return this.metrec;}
    public boolean getOccupé() {return this.occupé;}
    public void setOccupé(boolean b) {this.occupé=b;
        if (!this.occupé) {
            this.nomloc=null;
            this.loyerm=0;
        }
    }
    public double getLoyerm() {return this.loyerm;}
    public void setLoyerm(double l) {this.loyerm=l;}
    public String toString() {
        String res="Superficie = " + this.metrec;
        if (this.occupé) {
          res+= ",\n occupé par "+this.nomloc+" payant un loyer de "
                +this.loyerm;
        }
        return res;
    }
}
```

.....

▶ 2. Écrivez une méthode `main` dans laquelle vous créez trois appartements de superficie 18, 30 et 50 m^2 dont un seul possède un locataire et un loyer lors de son initialisation. Quelque temps plus tard, un autre appartement se loue. Modélisez ceci dans la méthode `main`. Voici un affichage possible sur la sortie standard, que vous devez reproduire avec le code de votre méthode `main`.

```
Superficie = 18.0
Superficie = 30.0, occupé par Marie payant un loyer de 1500.0
Superficie = 18.0, occupé par Guillaume payant un loyer de 600.0
```

```
public static void main(String[] args) {
    Appartement a1=new Appartement(18);
    System.out.println(a1);
    Appartement a2=new Appartement(30,"Marie", 1500);
    System.out.println(a2);
    Appartement a3= new Appartement(50);
    a1.setOccupé(true);
    a1.setNomloc("Guillaume");
```

```
    a1.setLoyerm(600);
    System.out.println(a1);
}
```

.....

Immeuble

On veut définir maintenant une classe `Immeuble`. Tout immeuble contient un nombre d'appartements fixe égal à 3. Cet ensemble d'appartements est défini à l'aide d'un tableau d'objets de type `Appartement`. Le nombre d'appartements est fixé à la création de l'immeuble (dans le constructeur). Une méthode :

```
public void ajoutAppartements(Appartement un,
                              Appartement deux,
                              Appartement trois)
```

permet d'ajouter la description des 3 appartements de l'immeuble. Mais, ces appartements ne sont pas forcément tous occupés, donc le loyer correspondant ne peut être perçu. Par conséquent, quand un appartement n'est pas occupé, il est représenté dans le tableau, mais on n'a pas de loyer ni de locataire associé.

La classe `Immeuble` devra posséder la méthode `surfacetotale` permettant de connaître la surface totale habitable, ainsi que la méthode `nbAppartOccupés` permettant de savoir combien d'appartements sont actuellement occupés dans l'immeuble.

▶ 3. Écrivez la classe `Immeuble` qui possède les propriétés données ci-dessus.

```
public class Immeuble {
    Appartement[] imm;
    public Immeuble() {
        imm=new Appartement[3];
    }
    public void ajoutAppartements(Appartement un,Appartement deux,
                                  Appartement trois)
    {
        imm[0]=un;
        imm[1]=deux;
        imm[2]=trois;
    }
    public double surfacetotale() {
        double surf=0;
        for (Appartement a: imm)
            surf+=a.getMetrec();
        return surf;
    }
    public int nbAppartOccupés() {
        int nb=0;
```



```
        for (Appartement a: imm) {
            if (a.getOccupé())
                nb++;
        }
        return nb;
    }
}
```

.....

▶ 4. Ajoutez dans la méthode `main` la création d'un immeuble. Il doit contenir les 3 appartements créés précédemment après qu'un second appartement ait été loué. Faites en sorte d'afficher sur la sortie standard la surface totale habitable de l'immeuble. Et pour finir, le nombre d'appartements actuellement occupés. L'affichage obtenu aura la forme :

```
Ce qui donne:
Surface totale habitable : 98.0
Nb apparts occupés : 2
```

```
Immeuble im1=new Immeuble();
im1.ajoutAppartements(a1, a2, a3);
System.out.println("Surface totale habitable :" +im1.surfacetotale());
System.out.println("Nb apparts occupés : "+ im1.nbAppartOccupés());
```

.....

Partie 2. Héritage

Dans cette partie, vous répondrez exactement aux questions posées, sans ajouter plus de code JAVA que celui qui est demandé.

▶ 5. Écrivez la classe `A` qui possède une variable (ou attribut) privée `a` de type **int** et un constructeur pour initialiser `a`.

```
class A {
    private int a;

    public A(int a) {
        this.a = a;
    }
}
```

.....

▶ 6. Écrivez la classe `B` qui hérite de `A` et qui possède une variable (ou attribut) privée `b` de type **int** et un constructeur pour initialiser `b`.

```
class B extends A {
    private int b;

    public B(int b) {
        this.b = b;
    }
}
```

.....

▶ 7. La compilation de ces deux classes provoque une erreur. Laquelle ? Expliquez ce qu'il faut ajouter dans la classe `A` pour corriger l'erreur.

Par défaut, le constructeur de `B` appelle le constructeur par défaut de `A` qui n'existe pas ! Pour corriger l'erreur, il suffit d'ajouter un constructeur par défaut dans `A`.

.....

▶ 8. On considère maintenant que la classe `A` est corrigée. Déclarez une classe `C` qui hérite de `A`.

```
class C extends A {}
```


.....

▶ 9. Dans une méthode `main` d'une classe de test, déclarez une variable `x` de type `A` initialisée avec un objet de type `B`. Puis, déclarez une variable `y` de type `A` initialisée avec un objet de type `C`. Enfin, déclarez une variable `z` de type `B`.

```
A x = new B(1);
A y = new C();
B z;
```

.....

▶ 10. Parmi les affectations suivantes, quelles sont celles qui sont correctes et qui sont incorrectes ? Expliquez pourquoi ? Comment corriger la ou les erreurs (s'il y en a) ?

```
x = y;
y = x;
z = x;
z = y;
```

Les deux premières affectations sont correctes puisque `x` et `y` sont de même type. Les deux dernières affectations sont incorrectes et le compilateur JAVA signale une erreur car un `A` est pas (nécessairement) un `B`. Toutefois, ici à l'exécution, `x` désigne un `B` et la 3ème affectation peut être faite avec une conversion explicite : `z = (B) x;`. En revanche pour la 4ème affectation, la conversion n'est pas possible car `z` est un `B` et `y` : `C` est sans relation avec `B`.

.....

▶ 11. On considère maintenant que les classes `A` et `B` possèdent toutes les deux la méthode **public void** `m()` qui écrit sur la sortie standard la valeur de leur variable de classe. À partir des affectations précédentes, indiquez si l'instruction `y.m();` est valide ou pas. Dans les deux cas expliquez pourquoi, et si elle est valide, donnez le résultat affiché.

Cette affectation est valide. La liaison dynamique est en jeu. A l'exécution, la variable `y` désigne un objet de type `C`, initialisé par son constructeur par défaut qui a initialisé la variable `a` de `A` à 0. la classe `C` n'a pas redéfini la méthode `m`, c'est donc celle de `A` qui est exécutée, et qui écrit donc 0 sur la sortie standard.

.....

Partie 3. Fichiers

Fichier d'entiers

On rappelle que la classe `Fichier` (vue en TD) à la forme suivante :

```
public class Fichier {
  private String nomFich;
  public Fichier(String f) { this.nomFich = f;}
  //....
}
```

On veut compléter la classe `Fichier` avec la méthode `sousFichier` qui fabrique et renvoie un `Fichier` d'entiers (**int**) formé des n premiers éléments du fichier courant. Si n est supérieur au nombre d'éléments du fichier courant, le fichier renvoyé contiendra tous les entiers du fichier courant. On rappelle que les classes `DataInputStream` et `DataOutputStream` permettent de traiter des fichiers binaires (`FileInputStream` et `FileOutputStream`) , et les méthodes `readInt` et `writeInt` permettent de lire et d'écrire des entiers de type **int**.

▶ 12. Écrivez la méthode `sousFichier` qui possède l'en-tête ci-dessous. Vous gérerez les exceptions nécessaires.

```
/*
 * Rôle : renvoie un Fichier de nom f construit à partir des (au plus)
 *        n premiers entiers de this
 */
public Fichier sousFichier(String f, int n)
```

```
public Fichier sousFichier(String f, int n) {
    try (DataInputStream din =
            new DataInputStream(new FileInputStream(this.nomFich));
         DataOutputStream dout =
            new DataOutputStream(new FileOutputStream(f)))
    {
       int i=0;
       while (i++<n) dout.writeInt(din.readInt());
    }
    catch (EOFException e) {}
    catch (IOException e) { System.err.println("Erreur d'E/S"); }
    finally { return new Fichier(f); }
}
```

.....

Fichier de texte

Un fichier de texte appelé `Notes` contient une suite de noms d'élèves, chaque nom étant lui-même suivi d'un réel quelconque mais positif de type double qui lui est associé et qui représente une note. Par exemple :

```
Pierre 9.9 Paul 10.89 Isabelle 56.90
Ali 30.60 Ran 280.12
Jacques 12.33 Carine 250.45
```

▶ 13. À l'aide de la classe `Scanner`, écrivez un programme qui lit le fichier `Notes` et qui écrit sur la sortie standard le nom de l'élève qui a la note la plus élevée. On considère que le fichier `Notes` est sans erreur, et que toutes les notes sont différentes. On rappelle que la classe `Scanner` possède un constructeur qui prend en paramètre un objet de `File` pour accéder au fichier à lire dont le nom sous forme de `String` est passé en paramètre à File. Elle possède également les méthodes `hasNext`, `next` et `nextDouble`, qui permettent respectivement, de vérifier s'il existe un prochain item à lire, de lire le prochain item, et enfin de lire le prochain réel double.

```
public static void main(String [] args) throws IOException {
    Scanner sc = new Scanner(new File("Notes"));
    String nom, nomMax=null;
    double noteMax=0;
    while (sc.hasNext()) {
        String nom = sc.next();
        double note = sc.nextDouble();
        if (note>noteMax) {
            noteMax = note;
            nomMax = nom;
        }
    }
     sc.close();
     System.out.println(nomMax==null ? "Fichier vide" : nomMax);
}
```

.....