Introduction à la programmation objet – Partiel – Octobre 2022

Durée 2h. Notes personnelles et documents de cours autorisés.
Barème approximativement proportionnel au temps nécessaire à la résolution de chaque question.

1 Questions de cours

On considère la classe `Intervalle` dont le code est donné page 3.

1. Fichiers.
   (a) Donner un nom possible d'un fichier contenant cette classe.
   (b) Lors d'un développement en Java interviennent des fichiers portant l'extension `.class`. Comment sont-ils créés et que contiennent-ils ?
2. Contenu d'une classe.
   (a) Identifier les attributs, les constructeurs et les méthodes de la classe `Intervalle`.
   (b) Que signifie l'indication **`private`** à la ligne 2 ?
   (c) L'affectation de la ligne 19 est-elle autorisée ? Pourquoi ?
   (d) Expliquer la signification de chaque occurrence de **`this`** dans ce code. Lesquelles sont facultatives ?
3. Objets.
   (a) Que se passe-t-il lors de l'exécution de la ligne 16 ? Détailler les différents éléments qui interviennent.
   (b) Écrire un appel de méthode permettant de tester si l'intervalle `i1` contient l'entier `4`.
   (c) Après l'exécution des lignes 16 à 18, combien d'objets ont-ils été créés ?
4. À la ligne 20 quelle serait la valeur de chacune des expressions suivantes ?
   (a) `i1.b` (b) `i2.b` (c) `i3.a` (d) `i1 == i2` (e) `i1 == i3`
5. Interfaces.
   (a) Écrire une interface Java que la classe `Intervalle` réalise.
   (b) Que faudrait-il modifier à la classe `Intervalle` pour déclarer ce lien avec l'interface ?
6. Affichage.
   (a) Qu'est-ce qui s'affiche lors de l'exécution de la ligne 21 ?
   (b) Écrire ce qu'il faut ajouter à la classe `Intervalle` pour que l'exécution de la ligne `21` affiche la ligne

```
[3, 4[
```

2 Messages d'erreur

Pour chacune des erreurs suivantes, dire si elle intervient lors de la compilation ou lors de l'exécution d'un programme Java, et proposer un code susceptible de produire l'erreur.

1.
```
~/src/Main.java:5:10
java: cannot find symbol
  symbol:   method getChar(int)
  location: variable s of type java.lang.String
```

2.
```
Exception in thread "main" java.lang.NullPointerException:
  Cannot invoke "String.charAt(int)" because "s" is null
        at Main.main(Main.java:5)
```

3.
```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
  String index out of range: 12
        at java.base/java.lang.StringLatin1.charAt(StringLatin1.java:48)
        at java.base/java.lang.String.charAt(String.java:1515)
        at Main.main(Main.java:5)
```

3 Programmation

On s'intéresse à une plateforme qui permet à des voyageurs de réserver un canapé où passer la nuit chez des particuliers sympas. Vous trouverez à la page suivante des squelettes pour :

- une classe `Periode`, désignant des intervalles entre une `Date` de `debut` et une date de `fin`,
- une classe `Canapreteur`, désignant un utilisateur prêtant son canapé et enregistrant notamment un ensemble de périodes disponibles pour accueillir un voyageur,
- la classe principale `CanaPret`, qui recense les canaprêteurs et les réservations en cours, et permet de gérer les réservations,

et l'interface `Date` des structures utilisées pour représenter les dates. Vous trouverez également en page 4 un extrait de la documentation Java pour la classe `ArrayList`.

Classe `Periode`

1. Dans la classe `Periode`, on considère que la date `debut` est systématiquement avant la date `fin`. Que peut-on faire dans le code pour le garantir ?
2. Écrire une méthode **`boolean`** `contient(Periode p)`, qui renvoie **`true`** si la période courante contient la période `p`.

Classe `Reservation` Une réservation est définie par une période, et par le canaprêteur dont le canapé est réservé.

3. Définir une telle classe, avec des attributs, un constructeur **`public`** `Reservation(Canapreteur cp, Periode p)` et des *getters* permettant de récupérer les valeurs des attributs.

Classe `Canapreteur`

4. Compléter le code du constructeur `Canapreteur(String nom)`.
5. Écrire une méthode **`public void`** `ajoute(Periode p)` qui ajoute la période `p` aux disponibilités du canaprêteur.
6. Écrire une méthode **`public boolean`** `disponible(Periode p)` qui renvoie **`true`** si la période `p` est incluse dans l'une des périodes de disponibilité du canaprêteur (on ne cherche pas à savoir si la période `p` peut être couverte par une combinaison de plusieurs périodes disponibles).
7. Écrire une méthode **`public`** `Periode retire(Periode p)` qui soustrait aux disponibilités du canaprêteur une période `pr` qui contient `p`, et renvoie la période `pr` trouvée. Dans le cas où il n'existe pas de période `pr` contenant `p`, la méthode doit renvoyer le résultat qui vous semble le plus adapté.
8. Écrire une méthode **`public boolean`** `reserve(Periode p)` qui tente de réserver le canapé du canaprêteur courant pour la période `p`, et qui renvoie **`true`** en cas de succès. La réservation doit :
   - identifier dans les disponibilités du canaprêteur une période `pr` contenant `p`,
   - faire en sorte que `p` ne soit plus disponible,
   - faire en sorte que les parties de `pr` qui ne sont pas dans `p` restent disponibles.

   Indication : il peut être nécessaire d'ajouter quelque chose à la classe `Periode`. Le cas échéant, précisez ce que vous ajoutez.

Classe `CanaPret`

9. Écrire une méthode **`public`** `ArrayList<Canapreteur> canapreteursDisponibles(Periode p)` qui renvoie un tableau contenant l'ensemble des canaprêteurs disponibles pour la période `p`.
10. Écrire une méthode **`public`** `Reservation reserve(Periode p)` qui créer et enregistre une réservation chez un canaprêteur disponible pour la période `p`, et qui renvoie la réservation créée.
11. Écrire une méthode **`public void`** `annule(Reservation r)` qui annule la réservation `r` et rend la disponibilité correspondante au canaprêteur concerné.
12. (Bonus) Faire en sorte que lorsqu'une réservation est annulée, la période rendue soit fusionnée avec les périodes disponibles voisines s'il y en a. Cela peut nécessiter d'agir sur d'autres classes que la seule classe `Canapret`.

Annexe exercice 1

```java
public class Intervalle {
  private int a, b;
  public Intervalle(int a, int b) {
    this.a = a;
    this.b = b;
  }
  public Intervalle(int a) {
    this(a, a+1);
  }

  public boolean contient(int c) {
    return this.a <= c && c < this.b;
  }

  public static void main(String[] args) {
    Intervalle i1 = new Intervalle(3);
    Intervalle i2 = new Intervalle(3, 4);
    Intervalle i3 = i1;
    i3.b = 5;
    ...
    System.out.println(i2);
  }
}
```

Annexe exercice 3

```java
interface Date {
  /* renvoie true si la date courante est inférieure ou égale à autreDate */
  boolean avant(Date autreDate);
  /* renvoie true si la date courante est égale à autreDate */
  boolean egale(Date autreDate);
}
```

```java
public class Periode {
  private Date debut, fin;
  public Periode(Date d, Date f) {
    this.debut = d;
    this.fin = f;
  }
  ...
}
```

```java
import java.util.ArrayList;
public class Canapreteur {
  private String nom;
  private ArrayList<Periode> dispos;
  public Canapreteur(String nom) {
    ...
  }
  ...
}
```

```java
import java.util.ArrayList;
public class CanaPret {
  private ArrayList<Canapreteur> canapreteurs;
  private ArrayList<Reservation> reservations;
  ...
}
```

Extrait de la documentation de la classe `ArrayList<E>`

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including **`null`**. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in *amortized constant time*, that is, adding n elements requires $O(n)$ time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

Each `ArrayList` instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an `ArrayList`, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

An application can increase the capacity of an `ArrayList` instance before adding a large number of elements using the `ensureCapacity` operation. This may reduce the amount of incremental reallocation.

Méthode	Description
`ArrayList<E>()`	(Constructor) Constructs an empty list with an initial capacity of ten.
`ArrayList<E>(int initialCapacity)`	(Constructor) Constructs an empty list with the specified initial capacity.
`boolean add(E e)`	Appends the specified element to the end of this list. Returns `true`.
`void add(int index, E element)`	Inserts the specified element at the specified position in this list.
`void clear()`	Removes all of the elements from this list.
`Object clone()`	Returns a shallow copy of this ArrayList instance.
`boolean contains(Object o)`	Returns true if this list contains the specified element.
`void ensureCapacity(int minCapacity)`	Increases the capacity of this `ArrayList` instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
`E get(int index)`	Returns the element at the specified position in this list.
`int indexOf(Object o)`	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`boolean isEmpty()`	Returns true if this list contains no elements.
`Iterator<E> iterator()`	Returns an iterator over the elements in this list in proper sequence.
`E remove(int index)`	Removes the element at the specified position in this list. Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the list.
`boolean remove(Object o)`	Removes the first occurrence of the specified element from this list, if it is present. If the list does not contain the element, it is unchanged. Returns `true` if this list contained the specified element.
`E set(int index, E element)`	Replaces the element at the specified position in this list with the specified element. Returns the element previously at the specified position.
`int size()`	Returns the number of elements in this list.

4 Corrections

4.1 Questions de cours

1. Fichiers.
   (a) `Intervalle.java` *(seule possibilité)*
   (b) Créés par le compilateur à partir des fichiers `.java`. Contiennent du *bytecode* exécutable par la JVM. Format binaire, pas lisibles par un humain.
2. Contenu d'une classe.
   (a) Attributs : `a`, `b`. Constructeurs : `Intervalle` (lignes 3 et 7). Méthodes : `contient` et `main`.
   (b) Les attributs `a` et `b` ne seront pas directement accessibles depuis l'extérieur de la classe.
   (c) Oui, car on est bien *dans* la classe `Intervalle`.
   (d) Lignes 4, 5, 12 : objet courant. Ligne 8 : autre constructeur de la classe courante. On peut omettre les **`this`** de la ligne 12.
3. Objets.
   (a) Création en mémoire de la structure destinée à représenter l'objet (dans le tas), puis appel du constructeur `Intervalle(int)` avec le paramètre 3, qui lui-même appelle le constructeur `Intervalle(int, int)` avec les paramètres 3 et 4, qui initialise les deux attributs.
   (b) `i1.contient(4)`
   (c) 2 (et `i3` et `i1` désignent le même).
4. Ligne 20.
   (a) 5 (b) 4 (c) 3 (d) **`false`** (e) **`true`**
   (point clé : `i1` *et* `i3` *désignent physiquement le même objet, et* `i2` *est distinct)*
5. Interfaces.
   (a) On ne cite ni les attributs ni les constructeurs. La mention **`public`** est superflue *(mais pas fausse)*. On pourrait également nommer les paramètres des méthodes.

```
1  interface I {
2    boolean contient(int);
3    void main(String[]);
4  }
```

   (b) `class Intervalle implements I ...`
6. Affichage.
   (a) Quelque chose de la forme `Intervalle@xxxxxxxx` représentant l'adresse de l'objet dans le tas.
   (b)

```
1  public String toString() {
2    return "[" + this.a + ", " + this.b + "[";
3  }
```

4.2 Messages d'erreur

1. Erreur à la compilation (utilisation d'un mauvais nom de méthode).

```
1  public class Main {
2    public static void main(String[] args) {
3      String s = "Hello";
4      s.getChar(12);
5    }
6  }
```

(dans le code, on ne veut pas forcément le même numéro de ligne, ce qui compte est d'avoir une erreur de la bonne nature)

2. Erreur à l'exécution (accès à un objet non initialisé). *(le code suivant ne produit pas l'erreur demandée, car il génère avant cela une erreur à la compilation pour variable non initialisée, mais est admis dans le cadre de cet exercice)*

```
public class Main {
  public static void main(String[] args) {
    String s;
    s.charAt(12);
  }
}
```

(pour obtenir vraiment l'erreur : initialiser explicitement avec `null` *ou, plus crédible, passer par un tableau de chaînes ou un objet contenant une chaîne, dont on n'initialise pas le contenu)*

3. Erreur l'exécution (accès hors des limites d'une chaîne).

```
public class Main {
  public static void main(String[] args) {
    String s = "Hello";
    s.charAt(12);
  }
}
```

4.3 Programmation

1. On peut ajouter un test au début du constructeur. En cas de dates mal choisies, déclencher une erreur ou choisir arbitrairement une combinaison correcte.
2. Il suffit de comparer les dates de début et de fin. *Rappel : les* `this` *sont falcultatifs. Critères : méthode bien formée, bon type de retour, bonne logique.*

```
public boolean contient(Periode p) {
  return (this.debut.avant(p.debut) || this.debut.egale(p.debut))
      && (p.fin.avant(this.fin) || p.fin.egale(this.fin));
}
// variante plus compacte :
// return !p.debut.avant(this.debut) && !this.fin.avant(p.fin);
```

3. *Critères : attributs présents et privés, constructeur qui initialise les deux attributs, bonne signature des* getters.

```
public class Reservation {
  private Canapreteur preteur;
  private Periode periode;
  public Reservation(Canapreteur cp, Periode p) {
    this.preteur = cp;
    this.periode = p;
  }
  public Canapreteur getPreteur() { return this.preteur; }
  public Periode     getPeriode() { return this.periode; }
}
```

4. *Point principal : création d'un tableau (avec constructeur par défaut ou avec une capacité au choix).*

```
public Canapreteur(String nom) {
  this.nom = nom;
  this.dispos = new ArrayList<>();
}
```

5. *Critère : utilisation correcte d'une méthode d'un attribut de l'objet courant.*

```
public void ajoute(Periode p) {
  this.dispos.add(p);
}
```

6. *Critères : énumération bien formée, appel d'une méthode définie à une question précédente, bonne valeur renvoyée.*

```
1  public boolean disponible(Periode p) {
2    for (Periode pr : dispos) {
3      if (pr.contient(p)) return true;
4    }
5    return false;
6  }
```

7. *Critères : énumération, combinaison d'appels, bonne valeur renvoyée.*

```
1  public Periode retire(Periode p) {
2    for (Periode pr : dispos) {
3      if (pr.contient(p)) {
4        dispos.remove(pr);
5        return pr;
6      }
7    }
8    return null;
9  }
```

8. On ajoute des *getters* à la classe `Periode`. Algo : on retire l'éventuelle période `pr` trouvée contenant `p`, puis on ajoute les éventuelles périodes de `pr` situées strictement ou strictement après `p`. *Critères : sélection d'une bonne période, test du résultat de la sélection, retrait de la période sélectionnée, bonne définition des reliquats.*

```
1  public boolean reserve(Periode p) {
2    Periode pr = this.retire(p);
3    if (pr == null) return false;
4    if (pr.getDebut().avant(p.getDebut()))
5      this.ajoute(new Periode(pr.getDebut(), p.getDebut()));
6    if (p.getFin().avant(pr.getFin()))
7      this.ajoute(new Periode(p.getFin(), pr.getFin()));
8    return true;
9  }
```

9. *Critères : définition et remplissage du tableau, valeur renvoyée du bon type.*

```
1  public ArrayList<Canapreteur> canapreteursDisponibles(Periode p) {
2    ArrayList<Canapreteur> cps = new ArrayList<>();
3    for (Canapreteur cp : canapreteurs) {
4      if (cp.disponible(p)) cps.add(cp);
5    }
6    return cps;
7  }
```

10. On choisit un prêteur arbitraire parmi les disponibles (ici, le premier), et on réserve auprès de lui. Au passage : création d'une réservation à enregistrer dans le tableau idoine et à renvoyer. *Critères : traitement du cas de la liste vide, sélection d'un prêteur, création et enregistrement d'une réservation.*

```
1  public boolean reserve(Periode p) {
2    ArrayList<Canapreteur> cps = canapreteursDisponibles(p);
3    if (cps.isEmpty()) {
4      return null;
5    } else {
6      Canapreteur cp = cps.get(0);
7      cp.reserve(p);
8      Reservation r = new Reservation(cp, p);
9      reservations.add(r);
10     return r;
11   }
12 }
```

11. *Critères : supprimer de la liste des réservations, et action sur le prêteur.*

```
1   public void annule(Reservation r) {
2     reservations.remove(r);
3     r.getPreteur().ajoute(r.getPeriode());
4   }
```

12. Plutôt que d'appeler directement la méthode `ajoute` du canaprêteur, on peut créer (dans `Canapreteur`) une méthode `annule` qui fait l'ajout après avoir opéré la fusion (c'est-à-dire retiré les périodes disponibles adjacentes et étendu la période à ajouter en conséquence).

```
1   public void annule(Reservation r) {
2     Date d = r.getPeriode().getDebut();
3     Date f = r.getPeriode().getFin();
4     for (Periode p : dispos) {
5       if (p.getFin().egale(d)) {
6         dispos.retire(p);
7         d = p.getDebut();
8       }
9       if (p.getDebut().egale(f)) {
10        dispos.retire(p);
11        f = p.getFin();
12      }
13      dispos.ajoute(new Periode(d, f));
14    }
15  }
```