Introduction à la Programmation Objet

2h, aucun document autorisé
Le barème est donné à titre indicatif et pourra être modifié.

20 octobre 2021

Vous pouvez abréger des nom de méthodes ou de variable dans votre copie, tant que cela reste clair pour un humain (par exemple `tHSGlobal()` pour `trouverHighScoreGlobal()`. N'hésitez pas à accompagner votre code de commentaires. Si un élément vous manque pour répondre, précisez ce que vous cherchez et comment vous vous seriez débrouillés sur ordinateur.

1 Questions de compréhension de cours (6 points)

Soit le code suivant :

```
1  import java.util.ArrayList;
2
3  public class JeuDeSociete implements Oeuvre {
4      private String titre;
5      private String auteur;
6      private String illustrateur;
7      private int ageMin;
8      private ArrayList<Integer> notesSur5;
9
10     public JeuDeSociete(String titre, String auteur, String illustrateur, int ageMin) {
11         this.titre = titre;
12         this.auteur = auteur;
13         this.illustrateur = illustrateur;
14         this.ageMin = ageMin;
15         this.notesSur5 = new ArrayList<>();
16     }
17
18     @Override
19     public String getTitre() {
20         return titre;
21     }
22
23     @Override
24     public String getArtiste() {
25         return this.auteur + ", "+ this.illustrateur;
26     }
27
28     public int getAgeMin() {
29         return ageMin;
30     }
31
32     public void setAgeMin(int age){
33         this.ageMin = age;
34     }
35
36     public void noter(int noteSur5){
37         this.notesSur5.add(noteSur5);
```

```
38      }
39
40      public double noteMoyenne(){
41          if (notesSur5.isEmpty()){
42              return 0;
43          }
44
45          double res = 0;
46          for (int note : notesSur5){
47              res+= note;
48          }
49          return res/notesSur5.size();
50      }
51
52      @Override
53      public String toString() {
54          return titre + "␣de␣" + this.getArtiste() + ",␣" + ageMin +"ans␣et␣plus";
55      }
56
57      public static void main(String[] args) {
58          JeuDeSociete boonlake;
59          boonlake = new JeuDeSociete("Boonlake", "Pfister", "Franz", 14);
60          System.out.println(boonlake);
61          System.out.println(boonlake.noteMoyenne());
62          boonlake.noter(5);
63          boonlake.noter(3);
64          JeuDeSociete pareil = boonlake;
65          pareil.setAgeMin(18);
66          boonlake.setAgeMin(12);
67          System.out.println(pareil);
68      }
69  }
```

Question 1 Quel est le nom de la classe ? Le nom du fichier ?

Question 2 Combien d'attributs possède une instance une cette classe ? Combien de paramètres prend le constructeur ?

Question 3 Que signifie le mot-clé `public` ligne 36 ? Est-il nécessaire pour pouvoir appeler la méthode `noter(int noteSur5)` dans le main ?

Question 4 Que fait l'instruction `new JeuDeSociete("Boonlake", "Pfister", "Franz", 14);` ligne 59 ?

Question 5 À quoi l'instruction `this.notesSur5 = new ArrayList<>();` ligne 15 ?

Question 6 Ligne 41, y a-t-il des cas où l'instruction `notesSur5.isEmpty()` pourrait lever une exception ? Cela peut-il se produire avec le code actuel ?

Question 7 Que signifie `implementsOeuvre` ligne 3 ?

Question 8 Pourquoi y a-t-il un `@Override` ligne 18 ?

Question 9 Qu'affiche l'instruction `"System.out.println(pareil);` ligne 68 ?

Question 10 Modifier la méthode `public String toString()` pour inclure la note moyenne à la représentation de l'objet.

2 Codage 6 points

Dans cet exercice, on va traiter une file d'attente de client, qui veulent passer à l'un des postes disponible pour traiter leur problème (par exemple une hotline envoyant les appels vers différents techniciens). On ne gèrera pas ici l'interface utilisateur (comment les événements d'arrivée de clients ou de disponibilité des postes sont déclenchés), on n'utilisera qu'une suite d'instructions dans le main (fourni).

Vous allez pour cela devoir respecter le squelette ci-dessous et utiliser la classe `LinkedBlockingQueue<E>`, dont la javadoc vous est fournie en annexe.

```
1  import java.util.ArrayList;
2  import java.util.concurrent.LinkedBlockingQueue;
3
4  public class FileDAttente {
5      ArrayList<Poste> postesOuverts;
6      LinkedBlockingQueue<Client> clientsEnAttente;
7
8      public FileDAttente(int nbClientsMaxDansLaQueue){
9          //TODO
10     }
11
12     public FileDAttente(){
13         //TODO
14     }
15
16     public void ouvrePoste(Poste c){
17         //TODO
18     }
19
20     public void clientArrive(Client c){
21         //TODO
22     }
23
24     public void accueilleClient(Poste poste){
25         //TODO
26     }
27
28     public void accueilleClient(){
29         //TODO
30     }
31
32     public double estimationTempsAttente(){
33         //TODO
34     }
35
36     public static void main(String[] args) {
37         Poste p1 = new Poste(1);
38         Poste p2 = new Poste(2);
39         Poste p3 = new Poste(3);
40         Client c1 = new Client();
41         Client c2 = new Client();
```

```
42          Client c3 = new Client();
43          Client c4 = new Client();
44          Client c5 = new Client();
45
46          FileDAttente queue = new FileDAttente();
47          queue.ouvrePoste(p1);
48          queue.ouvrePoste(p2);
49          queue.accueilleClient();
50          queue.clientArrive(c1);
51          queue.clientArrive(c2);
52          queue.clientArrive(c3);
53          queue.clientArrive(c4);
54          queue.ouvrePoste(p3);
55          queue.accueilleClient();
56          p1.devientDisponible();
57          queue.accueilleClient();
58          queue.clientArrive(c5);
59          p2.devientDisponible();
60          p1.devientDisponible();
61          queue.accueilleClient();
62          p3.devientDisponible();
63          queue.accueilleClient();
64      }
65  }
```

Le code de la classe `Poste` est fourni :

```
1   public class Poste {
2       private int numero;
3       private boolean disponible;
4
5       public Poste(int numero){
6           this.numero = numero;
7           this.disponible = true;
8       }
9
10      public boolean isDisponible() {
11          return disponible;
12      }
13
14      public void prendClient(Client c){
15          if (this.disponible) {
16              this.disponible = false;
17              c.traiterDemande();
18          } else throw new IllegalStateException("Comptoir␣"+numero+"␣deja␣occupe");
19      }
20
21      public void devientDisponible(){
22          this.disponible = true;
23      }
24  }
```

On possède de plus une classe `Client`, qui contient deux méthodes :

- `public void traiterDemande()`
- `public int tempsTraitementEstime()` qui donne le temps estimé de traitement du dossier du client, en minutes.

Question 11 Écrire un constructeur de `FileDAttente` prenant en paramètre la longueur maximale de la queue autorisée. Il initialisera les attributs, et utilisera une file de taille adaptée.

Question 12 Écrire un constructeur de `FileDAttente` sans paramètre, utilisant 20 par défaut comme longueur de queue maximale. Vous devez faire appel au constructeur défini à la question précédente.

Question 13 Écrire la méthode `void ouvrePoste(Poste p)` qui :

— Rend disponible le poste p

— Ajoute le poste à la liste des postes ouverts pour la file d'attente, à condition que celui-ci n'y soit pas déjà.

Question 14 Écrire la méthode `void clientArrive(Client c)` qui ajoute le client à la file d'attente, si celle-ci n'est pas déjà pleine. Sinon, on affiche "Client refusé" sur la sortie standard.

Question 15 Écrire la méthode `void accueilleClient(Poste p)` qui tente d'envoyer un client vers le poste p. Il faut pour cela que le poste soit disponible et qu'un client attende. Si ce n'ets pas le cas, la méthode ne fait rien.

Question 16 Écrire la méthode `void accueilleClient(Poste p)` qui tente d'envoyer un client vers le premier (dans l'ordre dans lequel ils sont listés) poste disponible.

Question 17 Modifier (réécrire) la méthode `void clientArrive(Client c)` pour tenter d'accueillir un client après l'arrivée d'un client dans la file.

Question 18 Peut-on utiliser une boucle for each sur les objets de type `LinkedBlockingQueue<E>` ? Justifier.

Question 19 Écrire la méthode `double estimationTempsAttente()` qui donne une estimation du temps d'attente d'un éventuel nouvel arrivant dans la file en additionnant les temps de traitement estimé de chacun des clients de la file, le tout divisé par le nombre de postes ouverts au moment où l'on appelle la méthode, en minutes.

3 Création de classes (8 points)

Dans cet exercice nous allons traiter l'ébauche d'un programme permettant à des agences immobilière de recenser leurs bien en location, et aux utilisateurs de les rechercher. Nous n'allons ici ne nous intéresser qu'à la création et la manipulation des entités de base de ce programme, sans interaction utilisateur. On ignorera les cas particulier réels (par exemple, les adresse seront toujours composé d'un numéro, d'une dénomination de rue, d'un code postal et d'une ville).

On supposera une classe Ville fournie, comportant tout le nécessaire à son bon fonctionnement ainsi que :

— une redéfinition adaptée de la méthode `String toString()`

— une redéfinition adaptée de la méthode `boolean equals(Object o)`

— une méthode `ArrayList<Ville> proches(int km)` renvoyant une liste de toutes les autres villes se trouvant à moins de `km` km.

Vous allez devoir écrire les classes :

— Adresse

— Logement

— Agence

Question 20 Écrire une classe Adresse possédant les attributs et constructeurs nécessaires, ainsi qu'une redéfinition adaptée de la méthode `String toString()`.

Question 21 Écrire une classe Logement possédant des attributs permettant de représenter :

— son adresse

— le loyer réclamé (entier, indiqué en €)

— la surface en mètre carrés

— un indicateur permettant de savoir si le logement est disponible.

On veillera à utiliser des noms et visibilités adaptés et à définir le ou les constructeurs nécessaires à la création d'instance de cette classe.

Question 22 Écrire une méthode `boolean respecteCriteres(Integer loyerMax, Integer surfaceMin)` qui renvoie vrai s le loyer demandé est inférieur à loyerMin et la surface en mètre carrés supérieure à surfaceMin. On utilisera les valeurs null pour ignorer un critère (ou les deux).

Question 23 Redéfinir dans la classe Logement la méthode `String toString()` pour renvoyer une représentation contenant la disponibilité, la ville, le loyer et la surface, de la forme :

```
Logement disponible : Orsay, 500€, 41m2
```

Question 24 Écrire une classe Agence, où chaque instance possède un nom et une `ArrayList` de logements rattachés. Fournir un constructeur et une méthode permettant d'ajouter un logement.

Question 25 Redéfinir dans la classe Agence la méthode `String toString()`, qui affichera le nom de l'agence puis la liste de tous les logements gérés par celle-ci, selon la forme précédemment demandée.

Question 26 Écrire dans la classe Agence une méthode `ArrayList<Logement> recherche(Ville ville, Integer loyerMax, Integer surfaceMin )` qui renvoie les logements libres de la ville donnée en respectant les critères (la valeur null est passée pour ignorer le critère). Vous pouvez ajouter des méthodes à d'autres classes si nécessaire (écrivez-les, en indiquant clairement dans quelles classes elles se situent).

Question 27 Écrire dans la classe Agence une méthode `ArrayList<Logement> rechercheProche(Ville ville, int km, Integer loyerMax, Integer surfaceMin )` qui renvoie les logements libres respectant les critères, et se situant à au plus `km` km de la ville indiquée.

Question 28 Écrire une méthode statique permettant d'afficher une liste de logements. Pourquoi utilise-t-on une méthode statique ?

compact1, compact2, compact3

java.util.concurrent

Class LinkedBlockingQueue<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractQueue<E>
      java.util.concurrent.LinkedBlockingQueue<E>

Type Parameters:

`E - the type of elements held in this collection`

All Implemented Interfaces:

`Serializable, Iterable<E>, Collection<E>, BlockingQueue<E>, Queue<E>`

```
public class LinkedBlockingQueue<E>
extends AbstractQueue<E>
implements BlockingQueue<E>, Serializable
```

An optionally-bounded blocking queue based on linked nodes. This queue orders elements FIFO (first-in-first-out). The *head* of the queue is that element that has been on the queue the longest time. The *tail* of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue. Linked queues typically have higher throughput than array-based queues but less predictable performance in most concurrent applications.

The optional capacity bound constructor argument serves as a way to prevent excessive queue expansion. The capacity, if unspecified, is equal to `Integer.MAX_VALUE`. Linked nodes are dynamically created upon each insertion unless this would bring the queue above capacity.

This class and its iterator implement all of the *optional* methods of the `Collection` and `Iterator` interfaces.

This class is a member of the Java Collections Framework.

Since:

`1.5`

See Also:

`Serialized Form`

Constructor Summary

Constructors

Constructor and Description
`LinkedBlockingQueue()`

Creates a LinkedBlockingQueue with a capacity of **Integer.MAX_VALUE**.

LinkedBlockingQueue(**Collection**<? extends **E**> c)
Creates a LinkedBlockingQueue with a capacity of **Integer.MAX_VALUE**, initially containing the elements of the given collection, added in traversal order of the collection's iterator.

LinkedBlockingQueue(int capacity)
Creates a LinkedBlockingQueue with the given (fixed) capacity.

Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method and Description
void	**clear**() Atomically removes all of the elements from this queue.
boolean	**contains**(**Object** o) Returns true if this queue contains the specified element.
int	**drainTo**(**Collection**<? super **E**> c) Removes all available elements from this queue and adds them to the given collection.
int	**drainTo**(**Collection**<? super **E**> c, int maxElements) Removes at most the given number of available elements from this queue and adds them to the given collection.
Iterator<**E**>	**iterator**() Returns an iterator over the elements in this queue in proper sequence.
boolean	**offer**(**E** e) Inserts the specified element at the tail of this queue if it is possible to do so immediately without exceeding the queue's capacity, returning true upon success and false if this queue is full.
boolean	**offer**(**E** e, long timeout, **TimeUnit** unit) Inserts the specified element at the tail of this queue, waiting if necessary up to the specified wait time for space to become available.
E	**peek**() Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
E	**poll**() Retrieves and removes the head of this queue, or returns null if this queue is empty.
E	**poll**(long timeout, **TimeUnit** unit)

	Retrieves and removes the head of this queue, waiting up to the specified wait time if necessary for an element to become available.
void	**put**(**E** e) Inserts the specified element at the tail of this queue, waiting if necessary for space to become available.
int	**remainingCapacity**() Returns the number of additional elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking.
boolean	**remove**(**Object** o) Removes a single instance of the specified element from this queue, if it is present.
int	**size**() Returns the number of elements in this queue.
Spliterator<**E**>	**spliterator**() Returns a **Spliterator** over the elements in this queue.
E	**take**() Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.
Object[]	**toArray**() Returns an array containing all of the elements in this queue, in proper sequence.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this queue, in proper sequence; the runtime type of the returned array is that of the specified array.
String	**toString**() Returns a string representation of this collection.

Methods inherited from class java.util.AbstractQueue

add, addAll, element, remove

Methods inherited from class java.util.AbstractCollection

containsAll, isEmpty, removeAll, retainAll

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.concurrent.BlockingQueue

add

Methods inherited from interface java.util.Queue

element, remove

Methods inherited from interface java.util.Collection

addAll, containsAll, equals, hashCode, isEmpty, parallelStream, removeAll, removeIf, retainAll, stream

Methods inherited from interface java.lang.Iterable

forEach

java.util

Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

```
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.

Note that this implementation is not synchronized. If multiple threads access an `ArrayList` instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the `Collections.synchronizedList` method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

```
List list = Collections.synchronizedList(new ArrayList(...));
```

The iterators returned by this class's `iterator` and `listIterator` methods are *fail-fast*: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Collection`, `List`, `LinkedList`, `Vector`, Serialized Form

Constructor Summary

Constructors

Constructor and Description
ArrayList() Constructs an empty list with an initial capacity of ten.
ArrayList(**Collection**<? extends **E**> c) Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.
ArrayList(int initialCapacity) Constructs an empty list with the specified initial capacity.

Method Summary

Methods

Modifier and Type	Method and Description
boolean	**add**(**E** e) Appends the specified element to the end of this list.
void	**add**(int index, **E** element) Inserts the specified element at the specified position in this list.
boolean	**addAll**(**Collection**<? extends **E**> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
boolean	**addAll**(int index, **Collection**<? extends **E**> c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
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.
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.
int	**lastIndexOf**(**Object** o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
E	**remove**(int index) Removes the element at the specified position in this list.
boolean	**remove**(**Object** o) Removes the first occurrence of the specified element from this list, if it is present.
boolean	**removeAll**(**Collection**<?> c) Removes from this list all of its elements that are contained in the specified collection.
protected void	**removeRange**(int fromIndex, int toIndex) Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
boolean	**retainAll**(**Collection**<?> c) Retains only the elements in this list that are contained in the specified collection.
E	**set**(int index, **E** element)

	Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

Methods inherited from class java.util.AbstractList

equals, hashCode

Methods inherited from class java.util.AbstractCollection

containsAll, toString

Methods inherited from class java.lang.Object

finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.List

containsAll, equals, hashCode