Pratique de la programmation orientée-objet
Examen final

31 mai 2019

Indications :

- l'examen dure de 12h15 à 16h00,
- indiquez votre nom, prénom et numéro SCIPER ci-dessous et sur toutes les éventuelles feuilles additionnelles que vous rendriez,
- placez votre carte d'étudiant sur la table.

Seuls les documents suivants sont autorisés, en version papier uniquement :

- les notes de cours,
- les énoncés et les corrigés des séries d'exercices,
- une feuille de résumé de format A4 au maximum, ne couvrant que le cours.

Aucun document concernant le projet n'est autorisé !

Bon travail !

Nom : _____

Prénom : _____

SCIPER : _____

1 Itérateur de liste [30 points]

Les itérateurs représentés par l'interface `Iterator` de Java sont très basiques, car ils ne permettent d'itérer que dans un sens. Pour cette raison, les listes offrent également une autre sorte d'itérateur, représenté par une interface différente nommée `ListIterator`, qui autorise le parcours dans les deux sens.

Le but de cet exercice est d'écrire une version simplifiée d'un itérateur de liste permettant d'itérer sur les éléments de la classe `SArrayList` présentée au cours. Ces itérateurs de liste simplifiés sont représentés par l'interface suivante :

```
public interface SListIterator<E> {
  public boolean hasNext();
  public E next();
  public int nextIndex();
  public boolean hasPrevious();
  public E previous();
  public int previousIndex();
  public void set(E newValue);
}
```

Les méthodes `hasNext` et `next` se comportent exactement comme les méthodes du même nom de l'interface `Iterator` standard. Ainsi, `next` retourne le prochain élément de la liste, et avance en même temps l'itérateur, ou alors lève l'exception `NoSuchElementException` si l'itérateur est à la fin de la liste.

La méthode `nextIndex` retourne l'index de l'élément qui serait retourné par le prochain appel à `next`, ou la taille de la liste si l'itérateur se trouve en fin de liste.

Les méthodes `hasPrevious`, `previous` et `previousIndex` sont similaires aux méthodes `hasNext`, `next` et `nextIndex`, mais permettent à l'itérateur de se déplacer en sens inverse dans la liste.

La méthode `previousIndex` retourne l'index de l'élément qui serait retourné par le prochain appel à `previous`, ou -1 si l'itérateur se trouve en début de liste.

Finalement, la méthode `set` remplace l'élément retourné par le dernier appel à l'une des méthodes `next` ou `previous` avec la valeur qu'on lui passe en argument, ou lève `IllegalStateException` si aucun appel à `next` ou `previous` n'a encore été fait.

La classe `SArrayList` ci-dessous est identique à celle présentée au cours, mais possède une méthode supplémentaire, `listIterator`, qui retourne un itérateur de liste positionné de manière à ce que le premier appel à `next` retourne le premier élément de la liste.

```
public final class SArrayList<E> implements SList<E> {
  private int size = 0;
  private E[] array = (E[]) new Object[10];

  // … autres méthodes (isEmpty, size, add, etc.)

  public SListIterator<E> listIterator() { à faire }
}
```

Écrivez le corps de la méthode `listIterator`.

Réponse :

2 Sérialisation ASCII [70 points]

La sérialisation ASCII consiste à représenter différents types de valeurs Java (p.ex. des entiers ou des chaînes) sous forme de chaînes ne contenant que des caractères ASCII. Cette représentation doit être réversible, c-à-d qu'il doit être possible d'obtenir les valeurs originales depuis leur représentation ASCII, par désérialisation.

L'interface `AsciiSerde` ci-dessous représente un *serde* — néologisme signifiant *serializer/deserializer*, c-à-d un objet capable de sérialiser et désérialiser des valeurs d'un type donné. Cette interface est générique, et son paramètre de type représente le type des valeurs que le serde est capable de (dé)sérialiser.

```
public interface AsciiSerde<T> {
  public Set<Character> alphabet();
  public String serialize(T value);
  public T deserialize(String str);
}
```

La méthode `alphabet` retourne l'alphabet utilisé par le serde, qui est l'ensemble des caractères ASCII dont il peut avoir besoin pour représenter les valeurs qu'il (dé)sérialise.

La méthode `serialize` retourne la chaîne représentant la version sérialisée de son argument. Cette chaîne ne doit être composée que de caractères appartenant à l'alphabet du serde.

La méthode `deserialize` retourne la valeur dont la version sérialisée est la chaîne qu'on lui donne en argument, ou lève `IllegalArgumentException` si cette chaîne est invalide, p.ex. car elle contient des caractères n'appartenant pas à l'alphabet du serde.

Les méthodes `serialize` et `deserialize` sont inverses l'une de l'autre.

La classe `BoolSerde` ci-dessous est un exemple de serde capable de (dé)sérialiser des valeurs booléennes en représentant la valeur `true` par la chaîne `T` et la valeur `false` par la chaîne `F` :

```
public final class BoolSerde implements AsciiSerde<Boolean>{
  public Set<Character> alphabet() {
    return new HashSet<>(Arrays.asList('T', 'F'));
  }
  public String serialize(Boolean value) {
    return value ? "T" : "F";
  }
  public Boolean deserialize(String str) {
    switch (str) {
    case "T": return true;
    case "F": return false;
    default:  throw new IllegalArgumentException();
    }
  }
}
```

Pour mémoire, les caractères ASCII incluent 33 caractères dits « de contrôle », qui sont invisibles et représentent p.ex. les retours à la ligne, ainsi que l'espace et les 94 caractères visibles suivants, dits « imprimables » :

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_‘abcdefghijklmnopqrstuvwxyz{|}~
```

En Java, les caractères sont des entiers (non signés), et les caractères ASCII sont simplement ceux dont la valeur est inférieure à 128. Dès lors, la méthode Java ci-dessous retourne vrai ssi son argument est un caractère ASCII :

```
boolean isAscii(char c) {
  return c < 128;
}
```

Partie 1 [20 points] Écrivez une classe nommée `IntSerde` représentant un serde pour les entiers de 32 bits, sérialisés sous forme signée, en base 10. L'alphabet de ce serde est composé des chiffres de 0 à 9 et du signe moins (`-`).

La méthode `serialize` produit des chaînes aussi courtes que possibles, c-à-d n'incluant aucun zéro ou signe moins (`-`) superflu en tête.

La méthode `deserialize` lève `IllegalArgumentException` si la chaîne qu'on lui passe contient des caractères ne faisant pas partie de l'alphabet du serde, ou des zéros ou signe moins (`-`) superflus en tête.

Le test JUnit ci-dessous, qui doit s'exécuter sans erreur, illustre l'utilisation de ce serde :

```
AsciiSerde<Integer> s = new IntSerde();
assertEquals("-12", s.serialize(-12));
assertEquals(12, s.deserialize("12"));
assertThrows(IllegalArgumentException.class,
             () -> s.deserialize("-0"));
```

Notez que la méthode `startsWith` de `String`, décrite dans le formulaire en annexe, peut vous faciliter la tâche.

Réponse :

(suite à la page suivante)

Réponse (suite) :

Partie 2 [20 points] Toute classe implémentant l'interface `AsciiSerde` doit satisfaire au moins les conditions suivantes pour être un serde valide :

1. l'alphabet retourné par `alphabet` ne doit contenir que des caractères ASCII,
2. toute chaîne retournée par `serialize` doit être constituée uniquement de caractères de l'alphabet du serde.

Écrivez une classe nommée `CheckingSerde` qui vérifie qu'un serde existant satisfait bien ces conditions.

Le constructeur de `CheckingSerde` accepte un serde quelconque en argument, qui est le serde à vérifier, et lève `IllegalSerdeException` ssi l'alphabet de ce serde contient au moins un caractère non ASCII.

`IllegalSerdeException` est définie ainsi :

```
public final class IllegalSerdeException
  extends RuntimeException { }
```

Les méthodes `alphabet`, `serialize` et `deserialize` de `CheckingSerde` se comportent exactement comme celles du serde passé au constructeur, à une différence près : la méthode `serialize` lève `IllegalSerdeException` si la chaîne produite par le serde passé au constructeur contient au moins un caractère n'appartenant pas à l'alphabet du serde.

Le test JUnit ci-dessous, qui doit s'exécuter sans erreur, illustre comment le comportement du serde de valeurs booléennes présenté plus haut peut être vérifié au moyen de la classe `CheckingSerde` :

```
AsciiSerde<Boolean> s = new CheckingSerde<>(new BoolSerde());
assertEquals("T", s.serialize(true));
assertEquals(false, s.deserialize("F"));
```

Notez que comme `BoolSerde` est valide — dans le sens où son alphabet {`T`, `F`} est composé uniquement de caractères ASCII, et sa méthode `serialize` ne produit que des chaînes composées de caractères provenant de cet alphabet — le serde `s` ci-dessus ne lèvera jamais une `IllegalSerdeException`.

Vous pouvez utiliser la méthode `isAscii` présentée plus haut pour tester si un caractère est un caractère ASCII.

Réponse :

Partie 3 [28 points] Écrivez une classe nommée `MapSerde` représentant un serde pour les tables associatives. Cette classe est générique, et ses deux paramètres de type représentent le type des clefs et le type des valeurs des tables associatives acceptées par le serde. Son constructeur prend trois arguments : le serde pour les clefs de la table, le serde pour les valeurs de la table, et un caractère de séparation.

`MapSerde` sérialise les tables associatives en sérialisant successivement les paires clef/valeur, dans l'ordre d'itération. Les clefs et valeurs sérialisées de chaque paire, ainsi que les paires successives, sont séparées par le caractère de séparation.

Le constructeur lève `IllegalArgumentException` si le caractère de séparation n'est pas un caractère ASCII, ou s'il appartient à l'alphabet du serde des clefs ou à celui des valeurs.

La méthode `deserialize` lève `IllegalArgumentException` si le nombre de chaînes obtenues en découpant la chaîne qu'on lui fournit au moyen du caractère de séparation est impair.

Le test JUnit ci-dessous, qui doit s'exécuter sans erreur, illustre l'utilisation de ce serde :

```
Map<Integer, Boolean> m = new TreeMap<>();
m.put(-1, true);
m.put(2019, false);
AsciiSerde<Map<Integer, Boolean>> s =
  new MapSerde<>(new IntSerde(), new BoolSerde(), ';');
assertEquals("-1;T;2019;F", s.serialize(m));
assertEquals(m, s.deserialize("-1;T;2019;F"));
```

Réponse :

(suite à la page suivante)

Réponse (suite) :

Partie 4 [2 points] Nommez les patrons de conception utilisés par :

1. `CheckingSerde` : _____
2. `MapSerde` : _____

3 Algorithme MCTS [15 points]

Admettons qu'une partie de Jass se déroule entre trois joueurs humains (joueurs 1 à 3) et un joueur simulé au moyen de l'algorithme MCTS (joueur 4). La partie vient de commencer, l'atout est trèfle (♣) et durant le premier pli, les joueurs humains jouent les cartes suivantes, dans cet ordre :

1. joueur 1 : as de cœur (♡A),
2. joueur 2 : valet de cœur (♡J),
3. joueur 3 : dame de cœur (♡Q).

Le joueur 4 doit maintenant jouer, et a les cartes suivantes en main :

1. pique : huit, neuf, dix, dame et roi (♠8, ♠9, ♠10, ♠Q, ♠K),
2. cœur : dix et roi (♡10, ♡K),
3. trèfle (atout) : six et valet (♣6, ♣J).

D'après les règles, seules les 4 dernières cartes sont jouables (♡10, ♡K, ♣6, ♣J).

Partie 1 [12 points] La figure 1 montre la racine de l'arbre de jeu qui est construit initialement par l'algorithme MCTS pour décider laquelle des quatre cartes jouer.

Complétez cette image en dessinant les nœuds qui sont ajoutés à l'arbre durant les quatre premières itérations de l'algorithme MCTS, et les arêtes qui les lient à leur nœud parent.

Dessinez les nœuds de gauche à droite, dans l'ordre dans lequel ils sont ajoutés à l'arbre, en vous souvenant que les cartes sont ordonnées d'abord par couleur (♠, ♡, ♢, ♣) puis par rang (6 à A). Dans chaque nœud, écrivez la même information que celle montrée à la figure 1 pour la racine, à savoir :

1. en haut, le pli correspondant à l'état du nœud, sous la forme d'une séquence de cartes entre crochets (souvenez-vous que l'arbre ne contient *jamais* de nœud correspondant à un pli plein),
2. en bas à gauche, le nombre de points moyen attribué au nœud,
3. en bas à droite, entre parenthèses, le nombre de tours qui ont été simulés à partir de ce nœud.

Annotez de plus chaque arête avec la carte à laquelle elle correspond.

Lorsque vous simulez la fin des tours, faites l'hypothèse que les scores suivants sont obtenus par les deux équipes, donnés ici en fonction de la carte jouée par le joueur 4 :

Carte	♡10	♡K	♣6	♣J
Équipe 1	106	83	54	74
Équipe 2	51	74	103	83

Partie 2 [3 points] Si une itération supplémentaire de l'algorithme devait être exécutée :

1. Où serait ajouté le nœud suivant (c-à-d quel nœud serait son parent) ? _____

2. Quelle carte serait jouée pour atteindre ce nœud ? _____

3. Quel joueur jouerait cette carte (1 à 4) ? _____

[♡A, ♡J, ♡Q]
0.00 (0)

Fig. 1: Arbre de jeu (à compléter)

4 Liaisons [10 points]

Le code ci-dessous construit une partie d'une interface graphique JavaFX montrant un chiffre et son nom en français et en anglais :

```
IntegerProperty i = new SimpleIntegerProperty(1);
ObservableList<String> fr = observableArrayList();
ObservableMap<String, String> frToEn = observableHashMap();

fr.addAll(”zero”, ”un”, ”deux”, ”trois” /* … ”neuf” */);
frToEn.put(”zero”, ”zero”);
frToEn.put(”un”, ”one”);
// … et ainsi de suite pour ”deux”, ”trois”, …, ”neuf”

Text t = new Text();
```

Le chiffre est stocké dans la propriété `i`, tandis que `fr` est une liste observable (constante) des noms français des chiffres, et `frToEn` est une table associative observable (constante) qui associe les noms anglais des chiffres à leur nom français. Finalement, `t` est un nœud de type `Text` destiné à afficher le chiffre contenu dans `i` et ses noms français et anglais. Par exemple, lorsque la propriété `i` contient 1, `t` doit afficher la chaîne :

```
1 (français: un, anglais: one)
```

Le code ci-dessus doit encore être complété pour s'assurer que le texte de `t` soit mis à jour correctement lorsque le chiffre stocké dans `i` change. Cinq versions du code qui pourrait être ajouté après le précédent pour tenter de faire cela sont données ci-dessous. Chacune de ces cinq versions est du code Java valide qui peut être exécuté sans causer d'erreur, mais malheureusement toutes ne fonctionnent pas. Lisez-les attentivement, en sachant qu'en Java il est légal d'utiliser la séquence de formattage `%s` avec un argument entier, de sorte que `String.format(”x=%s”, 10)` produit la chaîne `x=12`.

```
// Version 1:
t.textProperty()
  .set(String.format(”%s (français: %s, anglais: %s)”,
             i,
             Bindings.valueAt(fr, i),
             Bindings.valueAt(frToEn,
                              Bindings.valueAt(fr, i))));
```

```
// Version 2:
t.textProperty()
  .bind(Bindings.format(”%s (français: %s, anglais: %s)”,
                        i,
                        fr.get(i.get()),
                        frToEn.get(fr.get(i.get()))));
```

```
// Version 3:
i.addListener((property, oldValue, newValue) -> {
    String frVersion = fr.get((int) newValue);
    t.setText(String.format("%s (français: %s, anglais: %s)",
                            newValue,
                            frVersion,
                            frToEn.get(frVersion)));
  });
```

```
// Version 4:
t.textProperty()
  .bind(Bindings.format("%s (français: %s, anglais: %s)",
                i,
                Bindings.valueAt(fr, i),
                frToEn.get(Bindings.valueAt(fr, i))));
```

```
// Version 5:
t.textProperty()
  .bind(Bindings.format("%s (français: %s, anglais: %s)",
                i,
                Bindings.valueAt(fr, i),
                Bindings.valueAt(frToEn,
                                 Bindings.valueAt(fr, i))));
```

Cochez ci-dessous les versions qui sont correctes, dans le sens où le texte de `t` est mis à jour avec la bonne chaîne chaque fois que le chiffre stocké dans `i` change. Il n'est pas nécessaire de justifier votre réponse.

- ☐ Version 1
- ☐ Version 2
- ☐ Version 3
- ☐ Version 4
- ☐ Version 5

Formulaire

Ce formulaire présente toutes les parties de la bibliothèque Java nécessaires à cet examen. De nombreuses méthodes ont été omises et les types parfois simplifiés.

Classe `String`

La classe `java.lang.String`, immuable, représente une chaîne de caractères.

```
public class String {
  // Retourne la chaîne composée uniquement du caractère c.
  static String valueOf(char c);

  // Découpe la chaîne au moyen du délimiteur d et retourne le tableau des parties.
  static String[] split(String d);

  // Retourne la longueur de la chaîne.
  int length();

  // Retourne le caractère à l'index i ou lève IndexOutOfBoudsException
  // s'il est invalide.
  char charAt(int i);

  // Retourne vrai ssi la chaîne commence avec le préfixe p.
  boolean startsWith(String p);
}
```

Classe `StringJoiner`

La classe `java.lang.StringJoiner` représente un « joigneur » de chaîne, qui permet de construire des chaînes de caractères constituées d'une séquence de chaînes séparées par un délimiteur et entourée d'un préfixe et d'un suffixe.

```
public class StringJoiner {
  // Construit un « joigneur » de chaînes avec le séparateur d et un préfixe
  // et un suffixe vides.
  StringJoiner(String d);

  // Ajoute la chaîne s à la séquence.
  void add(String s);

  // Retourne la chaîne constituée de la concaténation du prefixe, des chaînes
  // ajoutées jusqu'à présent et séparées par le séparateur, et du suffixe.
  String toString();
}
```

(suite à la page suivante)

Interface `Map`

L'interface `java.util.Map` représente une table associative. Elle est implémentée, entre autres, par les classes `HashMap` et `TreeMap`.

```
public interface Map<K, V> {
  // Associe la valeur v à la clef k.
  V put(K k, V v);

  // Retourne l'ensemble des paires clef/valeur.
  Set<Map.Entry<K, V>> entrySet();
}
```

Interface `Map.Entry`

L'interface `java.util.Map.Entry` représente une paire clef/valeur.

```
public interface Map.Entry<K, V> {
  // Retourne la clef de la paire.
  K getKey();

  // Retourne la valeur de la paire.
  V getValue();
}
```

Interface `Set`

L'interface `java.util.Set` représente un ensemble de valeurs. Elle est implémentée, entre autres, par les classes `HashSet` et `TreeSet`.

```
public interface Set<E> extends Iterable<E> {
  // Retourne vrai ssi l'ensemble contient l'élément e.
  boolean contains(E e);

  // Ajoute l'élément e à l'ensemble et retourne vrai ssi il a été modifié
  // en conséquence.
  boolean add(E e);

  // Ajoute tous les éléments de c à l'ensemble et retourne vrai ssi il a été modifié
  // en conséquence.
  public boolean addAll(Collection<E> c);

}
```

(suite à la page suivante)

Classe `Collections`

La classe `java.util.Collections`, non instanciable, contient des méthodes statiques travaillant sur les collections.

```
public class Collections {
  // Retourne une vue non modifiable sur l'ensemble s.
  public static <T> Set<T> unmodifiableSet(Set<T> s);
}
```

Classe `Integer`

La classe `java.lang.Integer` contient entre autres des méthodes statiques travaillant sur les entiers de type `int`.

```
public class Integer {
  // Retourne la représentation textuelle en base 10 de l'entier i, sans
  // signe plus (+) ou zéro superflu en tête.
  static String toString(int i);

  // Retourne l'entier dont la chaîne s est la représentation textuelle en base 10,
  // ou lève NumberFormatException si s ne représente pas un entier valide.
  // Accepte (et ignore) un signe plus (+) superflu en tête, de même qu'un nombre
  // quelconque de zéros superflus.
  static int parseInt(String s)
    throws NumberFormatException;
}
```