Pratique de la programmation orientée-objet
Examen final

5 juillet 2021

Indications :

— l'examen dure de 16h15 à 20h00,

— indiquez votre nom, prénom et numéro SCIPER ci-dessous et sur toutes les éventuelles feuilles additionnelles que vous rendriez,

— placez votre carte d'étudiant·e 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 Égalité de gares [18 points]

La classe `Station` ci-dessous est similaire à celle du projet tCHu, mais son constructeur est privé et une méthode statique nommée `of` est fournie pour créer des instances. Dans un soucis de simplicité, la validation des arguments et la méthode `toString` ont été omis.

```
public final class Station {
  private final int id;
  private final String name;
  public static Station of(int id, String name) {
    return new Station(id, name);
  }
  private Station(int id, String name) {this.id = id; this.name = name;}
  public int id() { return id; }
  public String name() { return name; }
}
```

Une caractéristique importante de cette classe est que ses instances sont comparées par référence. Cela signifie que deux instances différentes de `Station` ayant les mêmes attributs ne sont *pas* considérées comme égales par `equals`. Ainsi, le test JUnit suivant échoue :

```
assertEquals(Station.of(13, "Lausanne"), Station.of(13, "Lausanne"));
```

Dans le projet, nous avons pris soin de ne jamais créer deux instances de la classe `Station` ayant les mêmes attributs, car cela aurait causé des problèmes. Le but de cet exercice est d'explorer deux versions alternatives de la classe `Station` qui ne nous obligeraient pas à prendre de telles précautions.

Partie 1 [8 points] Montrez comment redéfinir les méthodes `hashCode` et `equals` de `Station` afin que ses instances soient comparées par structure et non par référence.

Réponse :

Partie 2 [10 points] Au lieu de redéfinir `hashCode` et `equals`, une autre solution au problème décrit ci-dessus est de rendre impossible la création de deux instances différentes de `Station` ayant les mêmes attributs.

Montrez comment récrire la méthode `of` de `Station` afin d'empêcher la création de deux instances ayant les mêmes attributs. Cela implique de faire en sorte que, chaque fois que cette méthode est appelée avec la même identité et le même nom, elle retourne le même objet. Avec cette version de `Station`, le test JUnit suivant devrait s'exécuter avec succès :

```
Station s1 = Station.of(13, "Lausanne");
Station s2 = Station.of(19, "Neuchâtel");
Station s3 = Station.of(13, "Lausanne");
assertTrue(s1 == s3 && s1.id() == 13 && s1.name().equals("Lausanne"));
```

En d'autres termes, l'objet retourné par le premier appel à `of` devrait être le même que celui retourné par le troisième appel.

Conseil : stockez *toutes* les instances de `Station` créées dans une table associative.

Réponse :

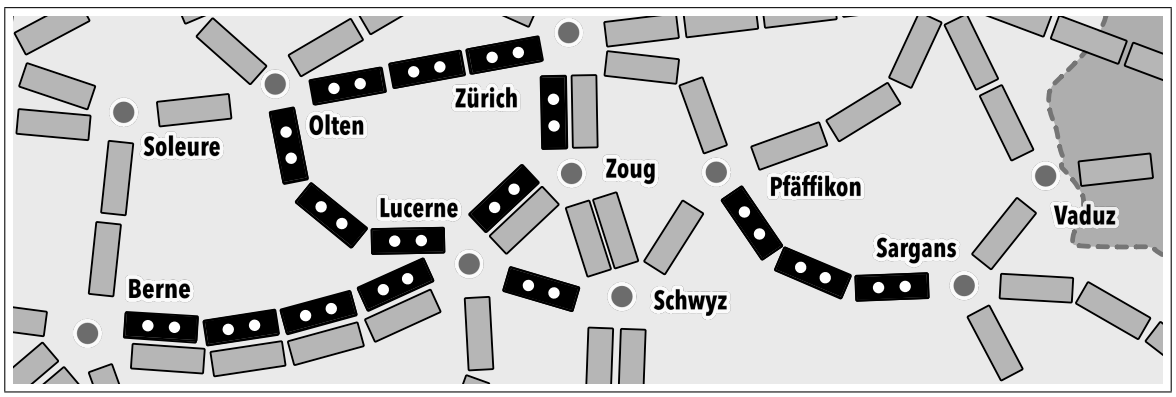

Fig. 1 : Extrait d'une carte de tCHu (utile aux exercices 2 et 4)

2 Partition de gares [16 points]

Dans certaines versions des *Aventuriers du Rail*, un bonus est accordé à la fin de la partie au joueur ayant connecté le plus grand nombre de gares en un réseau continu. En d'autres termes, ce bonus est attribué au joueur dont la partition de gares contient le plus grand sous-ensemble.

Par exemple, la carte de la figure 1 (page 3) montre que le plus grand nombre de villes que le joueur possédant les wagons noirs a réussi à connecter en un réseau continu est de 6. Il s'agit des gares d'Olten, Zürich, Berne, Lucerne, Zoug et Schwyz.

Le but de cet exercice est d'étendre la classe `StationPartition.Builder` du projet pour qu'elle calcule la taille du plus grand sous-ensemble de la partition qu'elle construit.

La classe `Builder` ci-dessous est la nôtre, présentée ici sans la classe `StationPartition` qui l'englobe, car celle-ci est inutile. La méthode `build` est encore incomplète.

```
public static final class Builder {
  private final int[] parent;
  public Builder(int stationCount) {
    parent = new int[stationCount];
    for (int i = 0; i < stationCount; i++) parent[i] = i;
  }
  public void connect(Station s1, Station s2) {
    parent[representative(s1.id())] = representative(s2.id());
  }
  public StationPartition build() {
    for (int i = 0; i < parent.length; i++)
      parent[i] = representative(i);
    int maxSubsetSize;        // à calculer !
    return new StationPartition(parent, maxSubsetSize);
  }
  private int representative(int n) {
    while (parent[n] != n) n = parent[n];
    return n;
  }
}
```

Récrivez le corps de la méthode `build` pour qu'elle calcule correctement la taille du plus grand sous-ensemble de la partition, et la stocke dans `maxSubsetSize`.

Réponse :

3 Ensembles de chaînes de caractères [80 points]

Un ensemble de chaînes de caractères peut être représenté par un tableau contenant les éléments triés par ordre croissant. Par exemple, l'ensemble de 5 chaînes de caractères {Java, Scala, Python, C, Rust} peut être représenté par un tableau contenant les éléments triés par ordre croissant, comme suit :

C	Java	Python	Rust	Scala

Pour tester la présence d'un élément dans un tel ensemble, on peut utiliser la recherche dichotomique, dont la complexité est de $O(\log n)$, où n est la taille de l'ensemble.

Le but de cet exercice est d'explorer cette idée en complétant la classe `StringSet` ci-dessous, qui représente un ensemble *immuable* de chaînes de caractères de cette façon. Les éléments de l'ensemble sont stockés par ordre croissant dans le tableau `elements`. Ce tableau est initialisé par le constructeur, qui est privé et ne valide ni ne copie le tableau qui lui est donné.

```
public final class StringSet implements Iterable<String> {
  private final String[] elements;
  private StringSet(String[] elements) { this.elements = elements; }
}
```

Attention : vous n'avez pas le droit d'utiliser une quelconque collection Java — c'est-à-dire une classe implémentant `List`, `Set` ou `Map` — pour cet exercice. Vous avez cependant le droit d'utiliser les méthodes de la classe `Arrays`, et avant d'aller plus loin, nous vous recommandons fortement de consulter la description de cette classe donnée dans le formulaire en annexe.

Partie 1 [8 points] Écrivez une méthode statique nommée `ofSortedArray` qui prend en argument un tableau de chaînes de type `String[]` et retourne une instance de `StringSet` contenant les mêmes éléments, ou lève une `IllegalArgumentException` si le tableau donné n'est pas trié par ordre croissant, ou contient des doublons.

Réponse :

Partie 2 [4 points] Écrivez les méthodes suivantes :

— `size`, qui ne prend aucun argument et retourne la taille de l'ensemble,

— `isEmpty`, qui ne prend aucun argument et retourne vrai ssi l'ensemble est vide.

Réponse :

Partie 3 [4 points] Écrivez une méthode nommée `contains`, qui prend en argument une chaîne de caractères et renvoie vrai ssi elle est contenue dans l'ensemble. Cette méthode *doit* effectuer une recherche dichotomique afin d'avoir une complexité en $O(\log n)$.

Réponse :

Partie 4 [24 points] Écrivez une méthode nommée `union`, qui prend en argument un autre ensemble de type `StringSet` et retourne l'union de cet ensemble et du récepteur (**`this`**), sous la forme d'un ensemble de type `StringSet`. Cette méthode doit avoir une complexité de $O(m+n)$, où m est la taille du récepteur, et n est la taille de l'argument.

Conseil : parcourez les tableaux des deux ensembles « en parallèle » afin de construire le tableau contenant leur union.

Réponse :

Réponse (suite) :

Partie 5 [8 points] Écrivez la méthode `iterator` — de l'interface `Iterable` —, qui retourne un itérateur itérant sur les éléments de l'ensemble dans l'ordre croissant. Comme l'ensemble est immuable, la méthode `remove` de cet itérateur doit simplement lever une exception de type `UnsupportedOperationException`.

Réponse :

Partie 6 [4 points] Écrivez une redéfinition de la méthode `toString` qui retourne une chaîne de caractères composée des éléments de l'ensemble séparés par une virgule et entourés de crochets. Par exemple, appliquée à l'ensemble donné ci-dessus, elle devrait retourner la chaîne `[C,Java,Python,Rust,Scala]`

Réponse :

Partie 7 [8 points] Pensez-vous que `StringSet` devrait redéfinir `hashCode` et `equals` afin que ses instances soient comparées par structure ? Si oui, écrivez leurs définitions. Sinon, expliquez pourquoi.

Réponse :

Partie 8 [2 points] La classe `StringSet` est déclarée ci-dessus comme suit :

```
public final class StringSet implements Iterable<String> { … }
```

et elle ne peut donc représenter que des ensembles de chaînes de caractères. Montrez comment vous adapteriez cette ligne pour déclarer une classe nommée `ComparableSet`, capable de représenter un ensemble de n'importe quelle classe implémentant l'interface `Comparable` de Java.

Réponse :

Partie 9 [18 points] Écrivez une classe imbriquée statiquement dans `StringSet`, nommée `Builder` et représentant un bâtisseur d'ensembles de chaînes de caractères, offrant les méthodes suivantes :

— `add`, qui prend une chaîne de caractères en argument, l'ajoute à l'ensemble en cours de construction puis retourne **`this`**,

— `compact`, qui ne prend aucun argument et compacte l'ensemble en cours de construction en triant ses éléments et en supprimant les doublons (voir ci-dessous) puis retourne **`this`**,

— `build`, qui ne prend aucun argument et retourne une instance de `StringSet` contenant les éléments ajoutés jusqu'à présent au bâtisseur.

Votre bâtisseur *doit* stocker les éléments de l'ensemble en cours de construction dans un unique tableau de type `String[]`, et le « redimensionner » (par copie) au besoin. La méthode `add` doit simplement ajouter le nouvel élément à ce tableau, sans chercher à éliminer les doublons, cette tâche étant laissée à `compact`.

Par exemple, après avoir exécuté l'extrait de programme suivant :

```
StringSet.Builder b = new StringSet.Builder()
  .add("Scala").add("Java").add("C").add("C").add("Scala") ;
```

le tableau contenu dans le bâtisseur `b` devrait contenir les éléments suivants, éventuellement suivis d'un nombre quelconque d'éléments nuls :

Scala	Java	C	C	Scala

Si `compact` est ensuite appelée sur le bâtisseur, son tableau doit être trié et les doublons éliminés, pour obtenir le tableau suivant (les deux derniers éléments sont nuls) :

C	Java	Scala	—	—

Notez que `compact` est censée être appelée à la fois explicitement par l'utilisateur du bâtisseur lorsque désiré, et par `build` pour obtenir le bon tableau à passer au constructeur (privé) de `StringSet`.

N'oubliez pas que vous n'avez pas le droit d'utiliser de collections pour cet exercice !

Réponse :

Réponse (suite) :

4 Chemins circulaires [18 points]

Dans certaines versions des *Aventuriers du Rail*, davantage de points sont accordés pour un billet ville à ville si le propriétaire de ce billet parvient à relier les deux villes par un chemin circulaire. C'est-à-dire, s'il existe un chemin constitué de routes appartenant au propriétaire du billet qui part de la première ville et passe par la seconde avant de revenir à la première.

Par exemple, dans la carte de la figure 1 (page 3), un billet Lucerne - Zürich appartenant au joueur possédant les wagons noirs vaudrait plus de points, car il existe un chemin composé de routes lui appartenant qui commence et finit à Lucerne et passe par Zürich : Lucerne - Zoug - Zürich - Olten - Lucerne. Par contre, un billet Berne - Schwyz appartenant au même joueur ne vaudrait pas plus de points que d'habitude, car il n'existe pas de chemin circulaire composé de routes lui appartenant qui commence à Berne et passe par Schwyz avant de revenir à Berne.

Le but de cet exercice est d'ajouter à la classe `Trail` du projet une nouvelle méthode qui trouve de tels chemins circulaires entre deux villes lorsqu'ils existent.

La déclaration de la classe `Trail`, de ses attributs et d'une méthode auxiliaire figure ci-dessous. Le constructeur (privé) et les accesseurs (publics), triviaux, ont été omis.

```
public final class Trail {
  private final Station station1, station2;
  private final List<Route> routes;
  private Trail tryToExtendWith(Route route) {
    for (Station s : route.stations()) {
      if (s.equals(station2)) {
        List<Route> extRoutes = new ArrayList<>(routes);
        extRoutes.add(route);
        return new Trail(station1, route.stationOpposite(s), extRoutes);
      }
    }
    return null;
  }
}
```

La méthode `tryToExtendWith` est utilisée par la méthode `longest` présentée plus loin, et vous sera également utile. Elle prend une route en argument et vérifie si le chemin auquel elle est appliquée (**`this`**) peut être étendu avec cette route. Si oui, le chemin étendu est retourné, sinon **`null`** est retourné.

Partie 1 [4 points] Écrivez le corps de la méthode privée suivante de `Trail`, qui renvoie vrai si et seulement si le chemin auquel elle est appliquée (**`this`**) est circulaire et passe par la station donnée (`via`) :

```
private boolean isCircularVia(Station via)
```

Par exemple, appliquée au chemin Lucerne - Zoug - Zürich - Olten - Lucerne avec Zürich comme argument, elle retournerait vrai.

Réponse :

Partie 2 [14 points] Écrivez le corps de la méthode publique et *statique* suivante de `Trail`, qui renvoie un chemin circulaire partant de — et donc se terminant à — la gare `start`, passant par la gare `via`, et composé uniquement de routes appartenant à la liste `routes` ; si un tel chemin n'existe pas, **`null`** est retourné.

```
static Trail circular(List<Route> routes, Station start, Station via)
```

Pour écrire cette méthode, vous pouvez utiliser un algorithme très similaire à celui utilisé par la méthode `longest` pour trouver le chemin le plus long. La principale différence est qu'au lieu de générer *tous* les chemins possibles, seuls ceux qui partent de la première gare (`start`) sont générés. Ensuite, dès que l'un d'entre eux est circulaire et passe par la deuxième gare (`via`), il est immédiatement retourné. Si aucun chemin de ce type ne peut être trouvé, **`null`** est retourné.

Pour vous aider à écrire la méthode `circular`, notre version de `longest` est donnée ci-dessous. Puisque `circular` et `longest` ont beaucoup de code en commun, vous pouvez vous référer directement aux lignes ci-dessous dans votre mise en œuvre de `circular` plutôt que de les copier. Par exemple, si vous voulez copier les lignes 3 à 6 (incluses) de `longest` dans votre code, écrivez simplement : « lignes 3 à 6 ».

```
1  public static Trail longest(List<Route> routes) {
2    List<Trail> trails = new ArrayList<>();
3    for (Route r : routes) {
4      trails.add(new Trail(r.station1(), r.station2(), List.of(r)));
5      trails.add(new Trail(r.station2(), r.station1(), List.of(r)));
6    }
7    Trail longest = new Trail(null, null, List.of());
8    while (!trails.isEmpty()) {
9      List<Trail> newTrails = new ArrayList<>();
10     for (Trail trail : trails) {
11       List<Route> unusedRoutes = new ArrayList<>(routes);
12       unusedRoutes.removeAll(trail.routes);
13       for (Route route : unusedRoutes) {
14         Trail extended = trail.tryToExtendWith(route);
15         if (extended != null) newTrails.add(extended);
16       }
17       if (trail.length() > longest.length()) longest = trail;
18     }
19     trails = newTrails;
20   }
21   return longest;
22 }
```

Réponse :

Réponse (suite) :

5 État immuable [18 points]

Les classes immuables `PublicGameState` et `GameState` ci-dessous représentent respectivement l'état public et l'état complet d'un jeu donné. Il s'agit de versions simplifiées des classes du même nom du projet tCHu.

La définition du type `Card` n'est pas donnée car elle n'a pas d'importance. Il pourrait p.ex. s'agir d'un type énuméré.

```
public class PublicGameState {
  private final int cardCount;
  public PublicGameState(int cardCount) { this.cardCount = cardCount; }
  public int cardCount() { return cardCount; }
}
public final class GameState extends PublicGameState {
  private final List<Card> cards;
  public GameState(List<Card> cards) {
    super(cards.size());
    this.cards = List.copyOf(cards);
  }
  public GameState withShuffledCards(Card c) {
    List<Card> newCards = new ArrayList<>(cards);
    Collections.shuffle(newCards);
    return new GameState(newCards);
  }
}
```

Cette manière de séparer les parties de l'état en deux classes distinctes, celle représentant l'état complet héritant de celle représentant l'état public, permet d'utiliser un état complet partout où un état public est attendu. Par exemple, il est possible de passer une instance de `GameState` à une méthode qui attend une valeur de type `PublicGameState`, ce qui est fort utile.

Cette solution a toutefois un inconvénient : `PublicGameState` ne peut pas être rendu finale, car cela empêcherait `GameState` d'en hériter. Ceci est regrettable, car les classes immuables devraient généralement être finales afin d'empêcher la définition de sous-classes non immuables.

Le but de cet exercice est de développer une solution alternative qui résout ce problème.

Partie 1 [6 points] Définissez deux *interfaces* nommées `PublicGameState` et `GameState` pour représenter respectivement la partie publique de l'état, et l'état complet du même jeu que ci-dessus. Ces interfaces doivent avoir exactement les mêmes méthodes que les classes plus haut, mais elles doivent toutes être abstraites. De plus, il devrait être possible d'utiliser une valeur de type `GameState` partout où une valeur de type `PublicGameState` est attendue.

Notez que ces interfaces sont destinées à remplacer complètement les classes ci-dessus, et le fait qu'elles aient le même nom n'a donc pas d'importance.

Réponse :

Partie 2 [6 points] Écrivez une classe *immuable* nommée `ConcretePublicGameState` qui implémente `PublicGameState` et fournit des versions concrètes de ses méthodes, ainsi qu'un constructeur prenant le même argument que celui de la classe `PublicGameState` ci-dessus.

Réponse :

Partie 3 [6 points] Écrivez une classe *immuable* nommée `ConcreteGameState` qui implémente `GameState` et fournit des versions concrètes de ses méthodes, ainsi qu'un constructeur prenant le même argument que celui de la classe `GameState` ci-dessus.

Réponse :

Page laissée intentionnellement vide.

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 `Arrays`

La classe `Arrays`, non instanciable, contient des méthodes statiques de manipulation de tableaux. (Voir aussi la classe `System` plus bas.)

```
public class Arrays {
  // Retourne une copie du tableau original de longeur newLength.
  // Les éventuels nouveaux éléments valent null.
  static <E> E[] copyOf(E[] original, int newLength);

  // Retourne l'index de l'élément key dans le tableau array, ou une valeur
  // (strictement) négative s'il ne s'y trouve pas.
  // La recherche est faite par dichotomie, en O(log n), et le tableau array
  // doit donc être trié par ordre croissant.
  static int binarySearch(Object[] array, Object key);

  // Place la valeur v dans tous les éléments du tableau array entre l'index
  // fromIndex (inclus) et l'index toIndex (exclus).
  static void fill(Object[] array, int fromIndex, int toIndex, Object v);

  // Retourne vrai ssi les tableaux a1 et a2 contiennent les mêmes éléments.
  static boolean equals(Object[] a1, Object[] a2);

  // Retourne une valeur de hachage basée sur le contenu du tableau array.
  static int hashCode(Object[] array);

  // Trie les éléments du tableau array selon leur ordre naturel. Ces éléments doivent
  // implémenter l'interface Comparable.
  static void sort(Object[] array);
}
```

Classe `System`

La classe `System`, non instanciable, contient diverses méthodes statiques.

```
public class Arrays {
  // Copie len éléments du tableau s, à partir de la position sPos, dans le tableau d,
  // à partir de la position dPos.
  void arraycopy(Object[] s, int sPos, Object[] d, int dPos, int len);
}
```

Classe `String`

La classe `String` représente une chaîne de caractères. Elle implémente l'interface `Comparable`, et possède donc une méthode `compareTo`.

(suite au verso)

Interface `Comparable`

L'interface `Comparable` est destinée à être implémentée par toutes les classes dont les instances peuvent être comparées (ordonnées) entre elles. De nombreuses classes de la bibliothèque Java l'implémentent, comme p.ex. `Integer`, `String`, etc.

```
public interface Comparable<T> {
  // Compare this et that et retourne un entier négatif si this < that,
  // zéro si this == that, et un entier positif sinon.
  int compareTo(T that);
}
```

Interface `Map`

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

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

  // Retourne la valeur associée à key, ou null s'il n'y en a aucune.
  V get(K key);
}
```

Classe `Objects`

La classe `Objects`, non instanciable, contient des méthodes utilitaires sur les objets.

```
public class Objects {
  // Retourne une valeur de hachage obtenue à partir de celles des objets objects.
  static int hash(Object... objects);
}
```

Classe `StringJoiner`

La classe `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 delimiter,
  // le préfixe prefix, et le suffixe suffix.
  StringJoiner(String delimiter, String prefix, String suffix);

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

  // 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();
}
```