

Examen (1h45, avec documents)

Nom :

Prénom :

- En cas de doute sur le sujet, notez vos choix sur la copie. Aucune réponse ne sera donnée par les surveillants.
- Il est conseillé de lire complètement le sujet avant de commencer à y répondre !
- Ne pas mettre les commentaires de documentation (sauf si nécessaires à la compréhension).

— Barème indicatif :

| | | | | | | |
|----------|---|-----|-----|-----|-----|-----|
| Exercice | 1 | 2.1 | 2.2 | 2.3 | 2.4 | 2.5 |
| Points | 6 | 1,5 | 2,5 | 3 | 2 | 5 |

Exercice 1 Répondre de manière concise et précise aux questions suivantes.

1.1 Indiquer ce que signifie l'annotation `@Override`, si elle est obligatoire et son intérêt.

1.2 La signature d'une méthode en Java peut intégrer une déclaration d'exceptions du type **throws** E. Indiquer quand ceci doit être utilisé et si ceci s'applique à toutes les exceptions.

1.3 On considère la signature suivante d'une méthode générique :

`<T> void m(Set<? super T> l1, Set<T> l2, Set<? extends T> l3);`

Expliquer ce que signifient les mots-clés **extends** et **super** et donner un exemple d'appel de cette méthode qui illustre l'intérêt de les avoir utilisés ici.

1.4 Dans l'API des collections Java, on trouve les notions de `List` et `ArrayList`. Comparer ces deux notions et indiquer quand les utiliser.

1.5 On considère le programme de test du listing 1 écrit en JUnit 4. Il compile et s'exécute sans erreur. La méthode `fail` de la classe `Assert` permet d'indiquer que le test échoue.

Pour cette question, il est conseillé de répondre directement sur le sujet.

1.5.1 Indiquer les éléments spécifiques de JUnit présents dans ce listing et les expliquer.

1.5.2 Indiquer et corriger les deux maladroresses importantes de cette classe de test.

1.6 Indiquer l'intérêt, pour un même concept, de définir à la fois une interface et une classe abstraite (par exemple `MouseListener` et `MouseAdapter` ou `List` et `AbstractList`).

Exercice 2 Dans cet exercice, nous modélisons une partie d'une application pour gérer le carburant mis dans un véhicule à moteur thermique.

2.1 Interface Plein. L'application gère les pleins qui sont faits sur un véhicule. Un plein est caractérisé par :

- l'odomètre (le nombre de kilomètres affiché au compteur du véhicule),
- le prix au litre du carburant,
- le prix payé pour le plein fait,
- la quantité de carburant ajouté dans le réservoir.

Listing 1 – Une classe de test

```
1 import org.junit.*;
2 import static org.junit.Assert.*;
3 import java.util.*;
4
5 public class SetTest {
6
7     @Test public void testAjoutElementPresent() {
8         Set<Integer> s = new HashSet<>();
9         Collections.addAll(s, 1, 2, 3, 4);
10        s.add(3);
11        if (s.size() != 4) {
12            fail("Ajouter_un_élément_présent_ne_change_pas_la_taille");
13        }
14    }
15
16    @Test public void testSuppressionElementAbsent() {
17        Set<Integer> s = new HashSet<>();
18        Collections.addAll(s, 1, 2, 3, 4);
19        s.remove(5);
20        if (s.size() != 4) {
21            fail("Supprimer_un_élément_absent_ne_change_pas_la_taille");
22        }
23    }
24
25    @Test public void testAjoutNouvelElement() {
26        Set<Integer> s = new HashSet<>();
27        Collections.addAll(s, 1, 2, 3, 4);
28        s.add(6);
29        if (s.size() != 5) {
30            fail("Ajouter_un_nouvel_élément_doit_augmenter_la_taille");
31        }
32    }
33
34 }
35 }
```

Par exemple, un plein peut être fait à 83 275 km pour ajouter 40,0 litres dans le réservoir pour un coût de 48,0 euros, le prix au litre étant de 1,20 euros.

2.1.1 Écrire en Java une interface qui décrit un plein. On ne donnera pas les commentaires de documentation. On ne définira aucune opération de modification d'un plein.

2.1.2 Expliquer comment il serait possible d'exprimer la relation qui lie le prix au litre, le prix payé et la quantité de carburant d'un plein.

2.2 *Classe PleinConcret.* On décide de stocker l'odomètre, le prix au litre et le prix payé.

2.2.1 Écrire la classe `PleinConcret` dont le constructeur prend en paramètre ces trois données.

2.2.2 Expliquer pourquoi il n'est pas possible de définir un second constructeur qui prend en paramètre l'odomètre, le prix au litre et la quantité et un troisième qui prend en paramètre l'odomètre, la quantité et le prix payé ?

2.2.3 Proposer une solution pour fournir ces trois manières de créer un plein.

2.3 *Véhicule.* On s'intéresse maintenant à la notion de véhicule qui répertorie tous les pleins réalisés pour un véhicule particulier.

2.3.1 Écrire une classe `Vehicule` qui propose une opération pour ajouter un nouveau plein. Cette classe doit être itérable sur les pleins.

2.3.2 Écrire dans une nouvelle classe `Outil` une méthode de classe qui, étant donné un véhicule, calcule le coût total dépensé en carburant.

2.4 *Diagramme de classe.* Dessiner le diagramme de classe qui fait apparaître `Plein`, `PleinConcret`, `Vehicule` et `Outil`.

2.5 *IHM en Swing.* On souhaite créer une interface utilisateur s'appuyant sur l'API Swing. Une première version est donnée au listing 2.

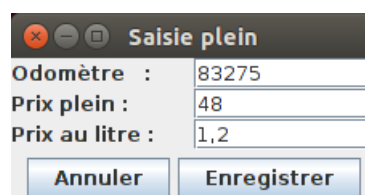
2.5.1 Dessiner la fenêtre qui s'affiche quand on exécute le programme du listing 2.

2.5.2 Modifier la classe du listing 2 pour obtenir l'interface utilisateur présentée à la figure 1.

2.5.3 Rendre actif le bouton Annuler (qui quitte l'application).

2.5.4 Rendre actif le bouton Enregistrer. Quand on clique sur "Enregistrer", un nouvel objet de type `Plein` doit être créé à partir des informations saisies si elles sont cohérentes. Si elles sont incohérentes, la zone de saisie correspondante aura son fond mis en rouge. Pour ce faire, on utilisera la méthode `setBackground` de `JTextField` qui prend en paramètre une couleur (`Color`).

On utilisera les méthodes `Double.parseDouble` et `Integer.parseInt` pour convertir une chaîne de caractères en réel ou entier. Elles lèvent l'exception `NumberFormatException` si la chaîne n'a pas le format attendu.



| Saisie plein | |
|--------------------------------|-------|
| Odomètre : | 83275 |
| Prix plein : | 48 |
| Prix au litre : | 1,2 |
| <div>Annuler Enregistrer</div> | |

FIGURE 1 – L'interface utilisateur souhaitée

Listing 2 – La classe PleinSwing

```
1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 public class PleinSwing {
6     final private JFrame fenetrePrincipale = new JFrame("Saisie_plein");
7     final private JTextField odometre = new JTextField(10);
8     final private JTextField prixAuLitre = new JTextField(10);
9     final private JTextField prix = new JTextField(10);
10    final private JButton annuler = new JButton("Annuler");
11    final private JButton enregistrer = new JButton("Enregistrer");
12
13    public PleinSwing() {
14        this.initComponents();
15        this.fenetrePrincipale.pack();
16        this.fenetrePrincipale.setVisible(true);
17    }
18
19    private void initComponents() {
20        JPanel p = new JPanel();
21        p.add(new JLabel("Odomètre_:_"));
22        p.add(this.odometre);
23        p.add(new JLabel("Prix_plein_:_"));
24        p.add(this.prix);
25        p.add(new JLabel("Prix_au_litre_:_"));
26        p.add(this.prixAuLitre);
27        p.add(this.annuler);
28        p.add(this.enregistrer);
29        this.fenetrePrincipale.getContentPane().add(p);
30    }
31
32
33    public static void main(String[] args) {
34        EventQueue.invokeLater(new Runnable() {
35            public void run() {
36                new PleinSwing();
37            }
38        });
39    }
40
41
42 }
```