Examen

– Il est conseillé de lire complètement le sujet avant de commencer à y répondre !
– Barème indicatif :

Exercice	1	2	3	4
Points	2	6	4	8

Exercice 1 Indiquer ce que nous apprend la trace d'exécution suivante.

```
[2]
Exception in thread "main" java.lang.ClassCastException: Attempt to insert class
    java.lang.Double element into collection with element type class java.lang.
    Integer
    at java.util.Collections$CheckedCollection.typeCheck(Collections.java:2276)
    at java.util.Collections$CheckedCollection.add(Collections.java:2319)
    at Main.main(Main.java:10)
```

1 Annuaire simplifié

Exercice 2 : Annuaire
L'objectif de cet exercice est de définir un annuaire simplifié qui permet de conserver des numéros de téléphone. L'interface annuaire (listing 1) spécifie les opérations de l'annuaire de manière minimaliste. Les commentaires de documentation sont incomplets mais un programme de test JUnit est donné (listing 2) pour en préciser le fonctionnement.

2.1 Expliquer ce qu'il faudrait ajouter à l'interface Annuaire pour qu'elle constitue réellement une spécification.

2.2 Indiquer ce qui justifie que la classe de test AnnuaireTest soit abstraite.

2.3 Expliquer ce que fait la méthode `getTel()` si le nom fourni ne permet pas de trouver de numéro de téléphone.

2.4 Expliquer l'information qu'il y a dans l'annuaire si on ajoute deux numéros de téléphone différents pour un même nom.

Listing 1 – L'interface Annuaire

```
public interface Annuaire {

   /** Enregistrer dans l'annuaire un nom avec son tél... */
   void ajouter(String nom, String tel);

   /** Obtenir le téléphone d'un contact à partir de son nom... */
   String getTel(String nom);
}
```

Listing 2 – La classe de test AnnuaireTest

```
1  import org.junit.*;
2  import static org.junit.Assert.*;
3
4  abstract public class AnnuaireTest {
5     protected Annuaire annuaire;
6
7     @Before public void setup() {
8        this.annuaire = newAnnuaire();
9        this.annuaire.ajouter("XC", "2186");
10       this.annuaire.ajouter("MP", "2185");
11    }
12
13    /** Retourner un annuaire vide. */
14    abstract protected Annuaire newAnnuaire();
15
16    @Test public void testerGetTel() {
17       assertEquals("2185", this.annuaire.getTel("MP"));
18       assertEquals("2186", this.annuaire.getTel("XC"));
19    }
20
21    @Test(expected=ContactInconnuException.class)
22    public void testerGetTelAbsent() {
23       this.annuaire.getTel("MO");
24    }
25
26    @Test public void testerAjouter() {
27       this.annuaire.ajouter("XC", "06...");
28       assertEquals("06...", annuaire.getTel("XC"));
29 }  }
```

2.5 Écrire la classe `ContactInconnuException`.

2.6 Écrire une réalisation de l'interface `Annuaire` appelée `AnnuaireConcret` en veillant à ce que :

1. l'opération `getTel(String nom)` soit efficace. Elle retourne le téléphone d'une personne à partir du nom de la personne.
2. la méthode toString() affichera l'annuaire dans l'ordre alphabétique des noms. Exemple :

```
MP 2185
XC 2186
```

On choisira avec soin les structures de données à utiliser !

2.7 `AnnuaireConcret` doit réussir les tests de `Annuaire`. Écrire la classe de test qui le fait.

2 Le patron de conception décorateur

L'objectif de ces exercices est d'étudier un patron de conception appelé « décorateur ». Nous l'appliquons sur un texte qui peut être affiché en gras ou en italique en utilisant les éléments HTML correspondants. Le principe d'HTML est d'utiliser des éléments délimités par une balise ouvrante et une balise fermante. Par exemple, pour mettre un texte en gras, on le met entre la balise ouvrante `<b>` et la balise fermante `</b>`. Ainsi `<b>gras</b>` affichera **gras**. Pour l'italique, on utilisera les balises `<i>` et `</i>` et pour le souligné, `<u>` et `</u>`.

Nous étudions une première modélisation et en identifions les défauts dans l'exercice 3. Nous appliquons alors le patron de conception « décorateur » dans l'exercice 4.

Exercice 3 : Texte formaté
Le diagramme de classe d'analyse [1] de la figure 1 propose une première modélisation de l'application avec une classe Texte et deux spécialisations.

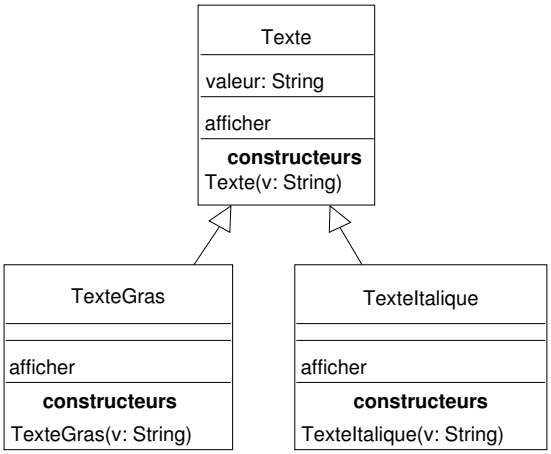

FIGURE 1 – Diagramme de classe des textes

3.1 Donner le code Java de la classe Texte.

3.2 Donner le code Java de la classe TexteGras.

3.3 Écrire un programme principal qui construit et affiche les trois textes suivants : 1) normal (donc `normal`) 2) **gras** (donc `<b>gras</b>`) et 3) *italique* (donc `<i>italique</i>`).

3.4 En conservant l'idée de cette modélisation, indiquer en complétant le diagramme de classe UML comment représenter : 1) un texte souligné et 2) un texte en gras et italique.

Exercice 4 : Décorateur
Le principal inconvénient de la modélisation précédente est qu'il est difficile de combiner plusieurs caractéristiques du texte. Il est par ailleurs impossible de les faire évoluer dynamiquement. Par exemple, un texte créé comme normal ne peut pas ensuite être considéré comme gras ou italique. Une nouvelle modélisation est proposée figure 2.

L'interface Texte a deux spécialisations TexteNormal et Format. TexteNormal correspond au texte de l'exercice précédent. Il s'agit d'une classe concrète. La classe Format est une classe abstraite qui fait référence à un Texte à travers le rôle texte. Il s'agit du décorateur. Sa méthode *afficher* est définie et consiste à appeler la méthode *afficher* du texte associé. Les classes Gras et Italique sont des sous-classes de Format. Elles correspondent à des formats (décorations) qui peuvent être ajoutés à un texte. Ces sous-classes redéfinissent la méthode *afficher*. Par exemple, la méthode *afficher* de Gras commence par afficher `<b>`, puis le texte, puis `</b>`.

1. Les deux premières rubriques des classes de ce diagramme correspondent respectivement aux requêtes et aux commandes.

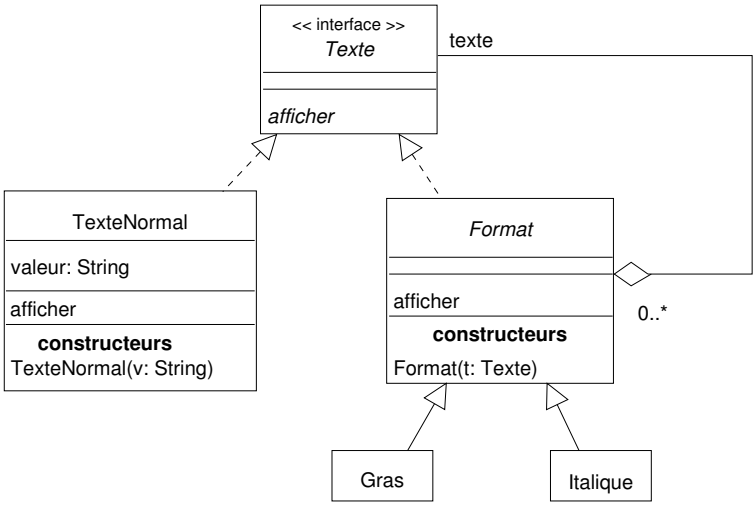

FIGURE 2 – Diagramme de classe avec le patron de conception décorateur

4.1 *Compréhension du décorateur.* Essayons de comprendre les choix faits par le concepteur de ce diagramme de classe.

4.1.1 Expliquer pourquoi Texte est définie comme une interface et Format comme une classe abstraite. On expliquera en particulier quels sont les inconvénients à utiliser une classe abstraite pour Texte ou une interface pour Format.

4.1.2 Le concepteur a défini un constructeur sur la classe abstraite Format. Indiquer s'il s'agit d'une erreur ou non.

4.1.3 Tous les éléments de la classe Format sont définis. Expliquer pourquoi le concepteur a décidé d'en faire une classe abstraite.

4.2 *Utilisation des classes.* Essayons maintenant de manipuler des textes.

4.2.1 Écrire une méthode de classe `enGrasEtItalique` qui prend en paramètre un Texte et retourne le même texte avec les décorations gras et italique ajoutées.

4.2.2 Dessiner le diagramme de séquence qui décrit ce qui se passe quand on applique la méthode `afficher` sur `enGrasEtItalique(new TexteNormal("OK"))`.

4.3 *Programmation en Java.* Intéressons nous maintenant au code.

4.3.1 Donner le code de l'interface Texte.

4.3.2 Donner le code de la classe abstraite Format.

4.3.3 Donner le code de la classe Gras.

4.4 *Extensions.* Envisageons quelques extensions de cette application.

4.4.1 Expliquer comment il serait possible d'ajouter la possiblité de souligner un texte.

4.4.2 On considère la notion de caractère (Caractere) qui peut aussi être mis en gras, italique ou être souligné. Expliquer sur le diagramme de classe comment prendre en compte cette notion.