Examen (1h30, sans documents)
Corrigé

- 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 (sur 20 points) :

Exercice	1	2	3
Points	8	6	6

Avertissement : Le corrigé donné ici n'est pas le seul possible. Tout ce qui est mis dans le corrigé n'est pas forcément attendu.

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

1.1 Pour écrire une exception `NouvelleException`, on hésite entre hériter de `RuntimeException`, de `Exception` ou de `Object`. Indiquer les conséquences à hériter de l'une ou l'autre de ces classes.

Solution :

Une exception doit être un sous-type de `Throwable`. On ne peut donc pas directement hériter de `Object`.

Si `NouvelleException` hérite de `Exception` elle sera vérifiée par le compilateur. Il faudra donc la spécifier grâce à **`throws`** dans la signature des méthodes qui la propagent (et le compilateur aidera le programmeur à ne pas oublier de la prendre en compte).

Si `NouvelleException` hérite de `RuntimeException`, elle sera non vérifiée par le compilateur, donc pas de **`throws`** à mettre mais pas d'aide non plus du compilateur. Ce sera au programmeur d'être vigilant et de penser à prendre en compte cette exception.

1.2 Comparer *interface* et *classe abstraite* : points communs et différences.

Solution : Classes abstraites et interfaces permettent de définir un comportement attendu. Dans les deux cas, elles ne peuvent pas être instanciées. Classes abstraites et interfaces ne pourront être utilisées que pour déclarer des poignées. L'intérêt est de pouvoir manipuler de manière uniforme des objets de types différents créés à partir de sous-types (sous-classes ou réalisations).

Les différences sont :

1. Les classes abstraites permettent de définir des attributs et des méthodes et ainsi de factoriser des éléments entre plusieurs classes alors qu'une interface ne peut que spécifier un comportement (depuis Java8, on peut aussi définir des méthodes statiques et des méthodes par défaut).
2. Les classes abstraites sont avant tout des classes et si on hérite d'une classe abstraite, on ne peut donc pas hériter d'une autre classe (héritage simple en Java pour les classes).

Au contraire, réaliser une interface n'empêche pas de réaliser d'autres interfaces (sous-typage multiple) et laisse la possibilité d'hériter d'une classe.

1.3 Définir et comparer *surcharge* et *redéfinition*.

Solution : La surcharge consiste à utiliser le même nom pour deux méthodes différentes qui doivent alors nécessairement avoir un nombre ou des types de paramètres différents. C'est le compilateur qui résout la surcharge en s'appuyant sur le type des poignées (types apparents).

La redéfinition est la possibilité pour une sous-classe de changer le code d'une méthode héritée. Les deux méthodes correspondent à la même opération et ont donc nécessairement la même signature. Le choix du code à exécuter est choisi à l'exécution en fonction du type de l'objet récepteur (type réel).

1.4 On considère la signature suivante d'une méthode `m` 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.

Solution :

`? super T` correspond à n'importe quel super-type de T.

`? extends T` correspond à n'importe quel sous-type de T.

Exemple :

```
1 Set<Object> so = ...;
2 Set<Point> sp = ...;
3 Set<PointNomme> spn = ...;
4 m(so, sp, spn);
```

Dans ce cas, T vaut Point (imposé par le deuxième paramètre) ; Object est bien un type plus général que T et PointNomme un type plus précis.

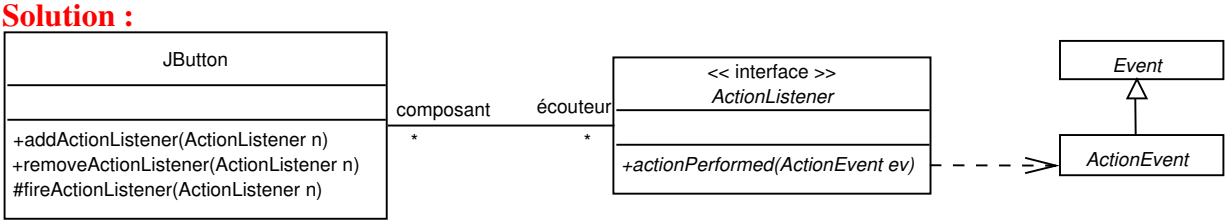
1.5 Expliquer ce qu'est un gestionnaire de placement dans l'API Swing.

Solution : Il s'agit d'un objet qui est associé à un container (par exemple un JPanel) et qui est responsable de gérer le placement des composants graphiques ajoutés à ce container.

Il y a plusieurs type de gestionnaire de placement qui correspondent à des politiques de placement différentes : FlowLayout, BorderLayout, GridLayout, etc.

1.6 Dessiner un diagramme de classe UML faisant apparaître les éléments suivants de l'API Swing : ActionListener, ActionEvent, Event, addActionListener, JButton et actionPerformed.

Solution :



ActionListener est l'interface qui spécifie une réaction qui pourra être enregistrée auprès d'un composant Swing tel qu'un bouton ou une entrée de menu, en fait un élement qui propose la méthode addActionListener. Cette méthode permet d'inscrire une nouvelle réaction auprès d'un composant. ActionListener définit une seule méthode actionPerformed qui correspond donc à la réaction et qui prend en paramètre un ActionEvent, objet qui transporte des informations

telles que le composant qui a produit l'événement (getSource), l'action associée (getActionCommand)... ActionEvent est une sous-classe de Event.

1 Questions à réponses courtes

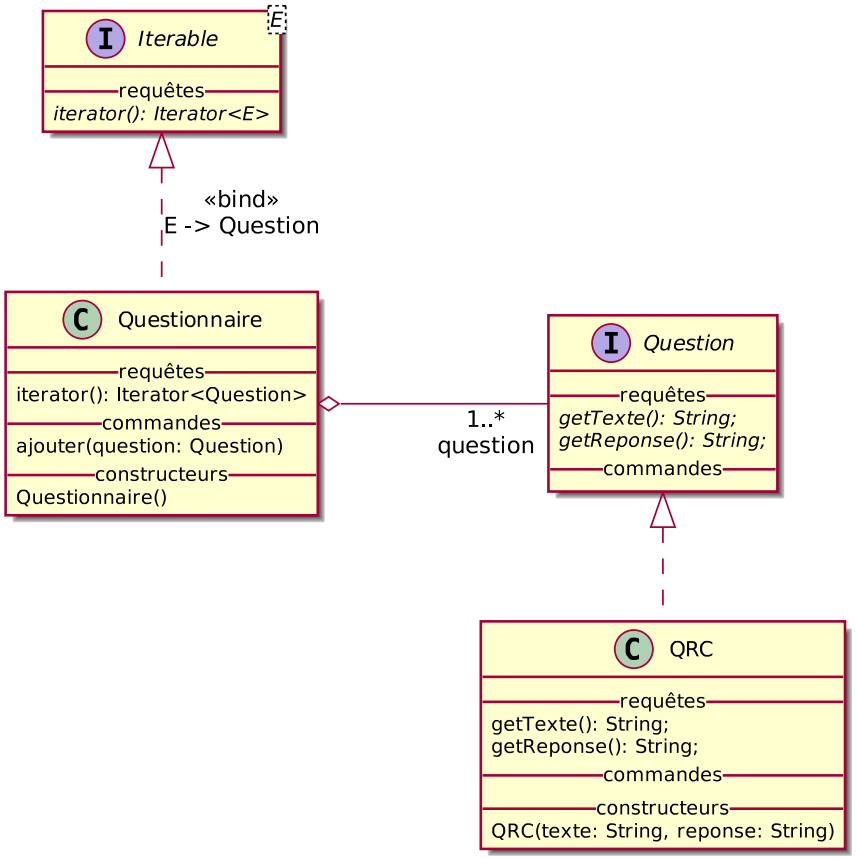
Exercice 2 : Questionnaire et questions à réponses courtes
On souhaite réaliser un questionnaire composé de plusieurs questions. On ne considèrera que des questions à réponse courte (QRC) mais on pourrait envisager d'autres types de questions comme par exemple des questions à choix multiple, des questions graduées, etc. Chaque question à réponse courte est caractérisée par le texte de la question (texte) et la réponse attendue (réponse).

On veut pouvoir itérer sur toutes les questions d'un questionnaire avec le `foreach` de Java comme dans l'exemple du listing 1.

Listing 1 – La classe ExempleQuestionnaire

```
1  public class ExempleQuestionnaire {
2     public static void main(String[] args) {
3        // Création d'un questionnaire
4        Questionnaire qTOB = new Questionnaire();
5        qTOB.ajouter(new QRC("Accessible à tous ?", "public"));
6        qTOB.ajouter(new QRC("Non modifiable ?", "final"));
7        qTOB.ajouter(new QRC("2 * 3 + 4 ?", "10"));
8
9        // Afficher le texte des questions
10       for (Question question : qTOB) {
11          System.out.println(question.getTexte());
12 }  }  }
```

2.1 Donner le diagramme de classe d'analyse qui fait apparaître `Question`, `QRC` et `Questionnaire` en utilisant les rubriques requêtes, commandes et constructeurs.
Solution :



Remarque : Iterable peut ne pas être représenté sur ce diagramme de classe.

2.2 Écrire `Question` (en Java).

Solution :

```java
public interface Question {
   String getTexte();
   String getReponse();
}
```

2.3 Écrire `Questionnaire` (en Java), en se limitant à ce qui est nécessaire à l'exécution du listing 1.

Solution :

```java
import java.util.*;

public class Questionnaire implements Iterable<Question> {

   private List<Question> questions;

   public Questionnaire() {
      this.questions = new ArrayList<>();
   }

   public void ajouter(Question question) {
      this.questions.add(question);
   }

   public Iterator<Question> iterator() {
      return this.questions.iterator();
} }
```

2.4 Écrire `QRC` (en Java).

Solution :

```
1  public class QRC implements Question {
2
3     private String texte;
4     private String reponse;
5
6     public QRC(String texte, String reponse) {
7        this.texte = texte;
8        this.reponse = reponse;
9     }
10
11    @Override
12    public String getTexte() {
13       return this.texte;
14    }
15
16    @Override
17    public String getReponse() {
18       return this.reponse;
19 }  }
```

Exercice 3 : Amélioration des QRC

Nous souhaitons apporter deux améliorations à nos QRC.

1. Une QRC peut avoir plusieurs réponses justes. La réponse attendue n'est donc pas unique.
2. Pour une QRC, il peut y avoir des erreurs fréquentes. Dans ce cas, on souhaite pouvoir expliquer à l'utilisateur l'erreur commise. Par exemple, pour la 3ème question, on peut lui dire « attention, la multiplication est prioritaire sur l'addition » s'il répond « 14 ». Bien sûr, il y autant de messages explicatifs que de mauvaises réponses classiques.

3.1 Pour la deuxième question du questionnaire du listing 1, les étudiants répondent souvent `static` au lieu de `final`. Quelle explication fournir, i.e. que signifie `static` en Java ?

Solution : `static` signifie qu'il s'agit d'un élément (attribut, méthode, classe, etc) attaché à un classe et non à un objet. On parle « de classe » par opposition à « d'instance » (attribut de classe ou attribut d'instance, méthode de classe ou méthode d'instance...).

3.2 Donner les attributs de `QRC` à ajouter ou modifier (en précisant leur type) pour prendre en compte ces évolutions.

Solution : Si plusieurs réponses sont possibles, il faut remplacer l'attribut `String reponse` par `Set<String> reponses`.

On utilise `Set<String>` car une même réponse ne peut apparaître qu'une seule fois et qu'il n'y a pas d'ordre sur les réponses.

Pour donner les indications, on peut ajouter un attribut de type tableau associatif :

```
Map<String, String> explications;
```

Le premier type String est la réponse erronée, la clé, et le second l'explication associée.

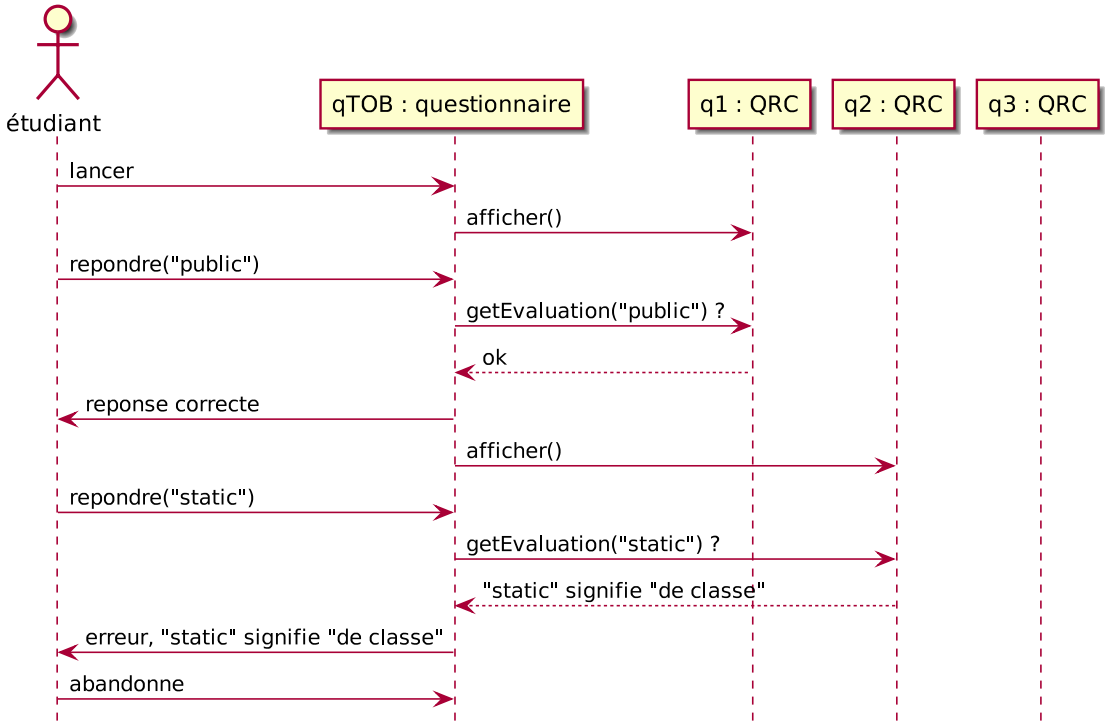
3.3 On considère le questionnaire du listing 1 qui contient 3 questions. Le scénario suivant décrit une utilisation possible de ce questionnaire, par exemple au travers de Moodle :

1. L'étudiant lance le questionnaire.
2. Le questionnaire affiche le texte de la première question.
3. L'étudiant propose "public".

4. Le questionnaire informe l'étudiant que la réponse est correcte
5. Le questionnaire affiche le texte de la deuxième question.
6. L'étudiant propose "static".
7. Le questionnaire informe l'étudiant que la réponse est incorrecte et lui affiche l'indication associée à "static" .
8. L'étudiant abandonne et retourne réviser son cours.

Dessiner un diagramme de séquence compatible avec ce scénario.

Solution :



3.4 D'après ce scénario, quelles méthodes faudrait-il ajouter sur `Questionnaire` et `Question` ?

Solution :

Sur `Questionnaire`, il faut prévoir :

— une méthode `lancer()` qui permet de démarrer le questionnaire,

— une méthode `proposer(texte: String)` qui permet à l'utilisateur de saisir sa réponse à la question posée (on pourrait utliser le terme « répondre » qui est aussi utilisé dans le sujet),

— une méthode pour `abandonner()` le questionnaire.

Sur `Question` (et donc `QRC`) :

— une méthode `afficher()` pour afficher le texte de la question.
On peut aussi se contenter de getTexte() pour récupérer le texte et c'est le questionnaire qui l'afficherait (plus logique, pour être plus indépendant de l'IHM).

— une méthode pour évaluer la réponse de l'utilisateur

```
getEvaluation(reponse: String): String
```

La chaîne de caractères pourra être "CORRECTE" si la réponse est correcte, sinon une explication est donnée (`null` si pas d'explication).
On pourrait faire deux méthodes pour éviter de surcharger le sens de la méthode `getEvaluation` :

— Une méthode pour savoir si la réponse est correcte `getEvaluation`.

```
estReponseCorrecte(reponse: String): boolean
```

— Une méthode pour obtenir l'explication en cas de réponse incorrecte.

```
getExplication(reponse: String): String
```

Remarque : La méthodes getReponse() aurait pu devenir getReponses() mais il est plus logique de définir une méthode estReponseValide(String) qui retourne un booléen. On garde ainsi les réponses cachées. C'est de la responsabilité de Question (ici QRC) de décider si la réponse est correcte ou pas. Dans la version originale, il aurait été préférable de faire la même chose !
Remarque : On pourrait prévoir une méthode :

```
void ajouterExplication(String reponse, String explication);
```

qui serait utilisée lors de la définition d'une QRC.