Projet de TP - Suite

NB : La lisibité du code et la réutilisation seront prises en considération dans la note du projet, ainsi que l'utilisation du polymorphisme.

Rappel de l'énoncé

On s'intéresse à la réalisation d'un jeu de culture générale sous-forme du jeu Pendu revisité, baptisé **EUREKA**. L'idée est de poser une question à l'utilisateur, puis, à lui de trouver la réponse et proposer la réponse en tapant des caractères dans une zone de saisie. Si un des caractères est faux, une partie d'un dessin de pendu est au fur et à mesure dessiné. En arrivant à 8 échecs (caractères saisis ne faisant pas partie de la réponse), le dessin du pendu est complété, puis la partie de jeu est terminée avec un échec.

Questions additionnelles

I. La liste des joueurs inscrits est sauvegardée dans un fichier **joueurs.txt**. La liste des questions également est sauvegardée dans un fichier **questions.txt.** Au lancement de l'application, celle-ci doit charger les structures HashMap et ArrayList respectivement des joueurs et questions par les informations contenues dans les fichiers de données.
Les instructions permettant de manipuler un fichier est donné en annexe.

II. On considère deux types de joueurs : Joueur Adulte et Joueur Enfant :
- A un joueur Adulte sera présentée une question pour Adulte,
- A un joueur Enfant sera présentée une question pour Enfant.

Pour cela, il est demandé de recourir à l'héritage et la notion de classe abstraite :

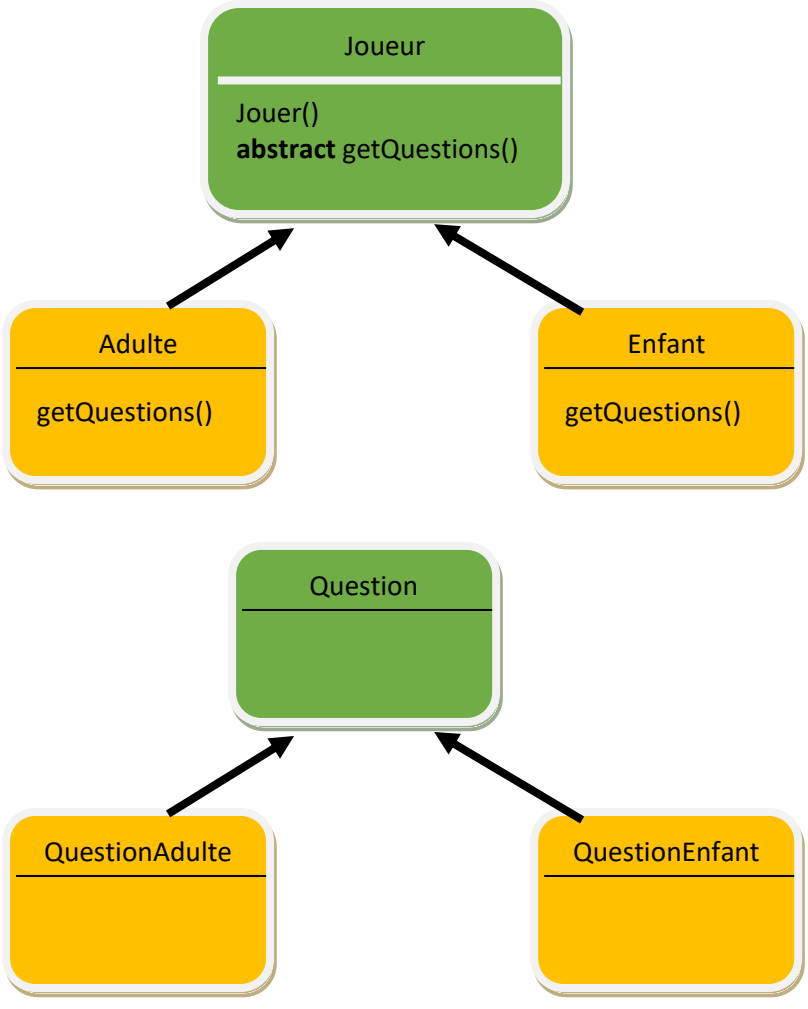
1. La classe Question va être transformée en une classe **mère** avec en introduisant deux classes filles **QuestionAdulte** et **QuestionEnfant**.
2. La classe Joueur va être transformée en une classe **abstraite** avec une méthode **Jouer**() et une méthode **abstraite ArrayList<Question> getQuestions(String thème)**.
3. Deux classes **Adulte** et **Enfant** vont être ajoutées comme classes filles de **Joueur**. Ces classes devront implémenter la méthode **getQuestions** telle que : pour l'adulte, la méthode **getQuestions** doit parcourir la liste des thèmes de l'application, chercher le thème demandé en paramètre et retourner uniquement les questions de type **QuestionAdulte** associées au thème; pour l'enfant, la méthode **getQuestions** doit parcourir la liste des thèmes de l'application, chercher le thème demandé en paramètre et retourner uniquement les questions de type **QuestionEnfant** associées au thème.
4. La méthode **Jouer**() de la classe Joueur va récupérer le thème sélectionné, puis appeler la méthode **getQuestions** avec comme paramètre le thème sélectionné, et récupère la liste des questions associées au joueur (soit adulte, soit enfant). Puis, il crée la fenêtre

qui affiche la question et tous les composants nécessaires à la potence (que vous avez déjà programmé).

Si vous voyez la nécessité d'ajouter des paramètres aux méthodes jouer() et getQuestions(…), vous pouvez le faire si vraiment nécessaire.

5. L'attribut Joueurs inscrits (**HashMap)** de votre application doit restée inchangée en contenant des Joueurs, qui peuvent être soient des adultes, soit des enfants.
6. A l'inscription d'un joueur, vous devez instancier soit un objet Adulte, ou bien un objet Enfant.
7. A la saisie ou récupération des questions et insertion dans la liste des questions d'un thème de l'application, il faudra instancier soit une question adulte ou bien une question enfant. Les objets contenus dans la liste des questions d'un thème ne doivent pas contenir des objets de type Question, mais soit **QuestionAdulte** , soit **QuestionEnfant**.
8. Ajouter dans la classe application une méthode **AfficherInfosJoueursInscrits**() qui parcourt la hashmap des joueurs et affiche les informations de tous les joueurs inscrits, avec leur type si adulte ou enfant. Vous pouvez pour cela redéfinir la méthode toString pour la classe **Joueur** ou bien les classes **Adulte** et **Enfant** (à vous de réfléchir). Ajouter un bouton permettant de visualiser la liste des joueurs inscrits sur la fenêtre principale de l'application, faisant appel à cette méthode.

Il est nécessaire d'utiliser la notion de polymorphisme ici.



1. Annexe : Lecture et écriture d'objets dans un fichier : la sérialisation

La **sérialisation** est un processus permettant de convertir des objets en un flux d'octets. Une fois convertis en flux d'octets, ces objets peuvent être écrits dans un fichier, ou envoyé via un réseau ou un protocole de communication donné. Le processus inverse de ceci s'appelle la **dé-sérialisation**.

Objet Java Serialization est une API fournie par la pile de la bibliothèque Java afin de sérialiser les objets Java.

Deux classes sont utilisées : **ObjectInputStream** et **ObjectOutputStream**, qui permettent de transférer les objets comme des séries d'octets, et donc de **sérialiser** les objets (opération de **sérialisation)**. Ces deux classes enregistrent les valeurs des attributs des objets, plus des informations supplémentaires sur la classe.

Pour qu'un objet soit **sérialisable** (çàd sur lequel on peut faire la sérialisation), il doit implémenter l'interface **Serializable** qui ne définit aucune méthode, mais autorise la sérialisation de l'objet.

Exemple

Person.java

```
package com.usthb;
import java.io.Serializable;

public class Person implements Serializable {
   private String name;
   private int age;
   private String gender;

   Person() {    };
   Person(String name, int age, String gender) {
      this.name = name;
      this.age = age;
      this.gender = gender;    }

   @Override
   public String toString() {
      return "Name:" + name + "\nAge: " + age + "\nGender: " + gender;    }
}
```

Les objets peuvent être convertis en flux d'octets à l'aide de java.io.ObjectOutputStream . Afin de permettre l'écriture d'objets dans un fichier en utilisant ObjectOutputStream , il est obligatoire que la classe concernée implémente l'interface` **Serializable**`.
Lire des objets en Java est similaire à écrire en utilisant ObjectOutputStream ObjectInputStream L'exemple ci-dessous montre le cycle complet d'écriture et de lecture d'objets en Java.
Lors de la lecture d'objets, ObjectInputStream essaie directement de mapper tous les attributs dans la classe dans laquelle nous essayons de convertir l'objet en lecture. S'il ne parvient pas à mapper l'objet correspondant exactement, il génère une exception ClassNotFound .

Exemple utilisant la classe Person précédente

WriterReader.java

```
package com.usthb;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class WriterReader {

   public static void main(String[]args) {
      Person p1 = new Person("Rédha", 30, "Male");
      Person p2 = new Person("Imane", 25, "Female");

      try {
         FileOutputStream f = new FileOutputStream(new File("myObjects.txt"));
         ObjectOutputStream o = new ObjectOutputStream(f);

        //Ecriture des objets sur un fichier
         o.writeObject(p1);
         o.writeObject(p2);
         //Fermeture
         o.close();
         f.close();

         FileInputStream fi = new FileInputStream(new File("myObjects.txt"));
         ObjectInputStream oi = new ObjectInputStream(fi);

        //Lecture des objets
         Person pr1 = (Person) oi.readObject();
         Person pr2 = (Person) oi.readObject();

         System.out.println(pr1.toString());
         System.out.println(pr2.toString());

         oi.close();
         fi.close();
      }
     catch (FileNotFoundException e) { System.out.println("File not found"); }
     catch (IOException e) { System.out.println("Error initializing stream"); }
     catch (ClassNotFoundException e) {  e.printStackTrace();       }
   }

}
```