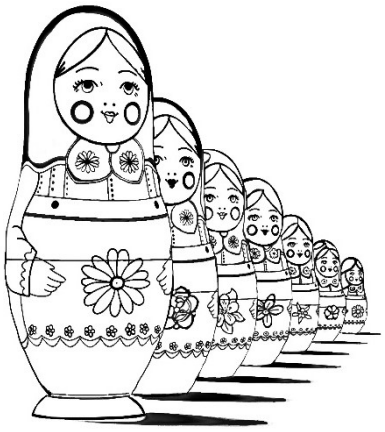
| | | |
|---|---|---|
| | Institut Supérieur des Etudes Technologiques de Sousse<br>Département de Technologies d'informatique<br>Licence appliquée en Technologies d'informatique<br>**DEVOIR SURVEILLE** | **Année universitaire**: 2023/2024<br>**Semestre**: 2<br>**Date**: Avril 2024<br>**Durée** : 1h |
| **Documents** : Non autorisés | **Unité d'enseignement** : Programmation orientée objet avancée<br>**Matière** : Programmation orientée objet avancée | **Classe** : DSI2.2<br>**Nb. Pages**: 2+1annexe |
| **Enseignant :** Jalel Boughizane | | |

## *Partie 1 : Java avancée (10pts)*

### A. Généricité et collections : Les poupées russes (4= 2 +2 points)



On souhaite modéliser le fonctionnement des poupées russes à l'aide d'une structure de données.

Pour cela on définit la classe **ListeDePoupees** qui hérite de la version générique de la classe **ArrayList**. Le paramètre générique de la classe **ListeDePoupees** doit être défini de manière qu'on n'autorise que des poupées Russes à être ajoutées à la liste.

***Question 1.*** **Ecrire l'entête (prototype) de la classe ListeDePoupees**

L'objectif est de modifier la méthode *add*() de la classe générique **ArrayList** de manière à réaliser les tests nécessaires avant l'ajout.

- Soit le code et l'entête de la méthode **add**() de la classe **ListeDePoupees** qui permet d'ajouter en fin de liste la poupée passée en paramètre si les conditions suivantes sont respectées :
  1. La dernière poupée de la liste est fermée et a une taille inférieure à la poupée passée en paramètre,
  2. La poupée passée en paramètre est ouverte et vide.
     - On considère la classe **PoupeeRusse** déjà codée et possédant les méthodes nécessaires à son utilisation. Par exemple la méthode **isFerme** permet de retourner la valeur true si la PoupeeRusse est fermé et la méthode **getTaille** permet de retourner la taille de la PoupeeRusse.

```
5   public boolean add(PoupeeRusse poupeeCourante) {
6       int count = this.size();
7       PoupeeRusse poupeFin = this.get(count);
8       if (poupeFin.isEtat() == true)
9           if (poupeFin.getTaille() < poupeFin.getTaille()) {
10              add(poupeeCourante);
11              return true;
12          }
13      return false;
14  }
```

***Question 2.*** **Préciser les erreurs (s'ils existent) dans le code précédent, corriger et donner le code correct**

## B. Expression lambda et Stream ( 6= 2+2+2 points)

Soit **listePoupeesRusse** une instance de type List<**PoupeeRusse**>.

***Question 3.*** **Expliquer chaque instruction et Préciser le résultat de chaque code .**

Code A

```
1. listePoupeesRusse.stream()
2.      .sorted((e1,e2) -> (int) (e1.getTaile() - e2.getgetTailee()))
3.      .forEach(e -> System.out.println( e.getId()));
```

Code B

```
1. listePoupeesRusse.stream()
2.       .filter(e -> e.isFerme() && e.getTaille() > 5)
3.       .collect(Collector.toList());
```

Code C

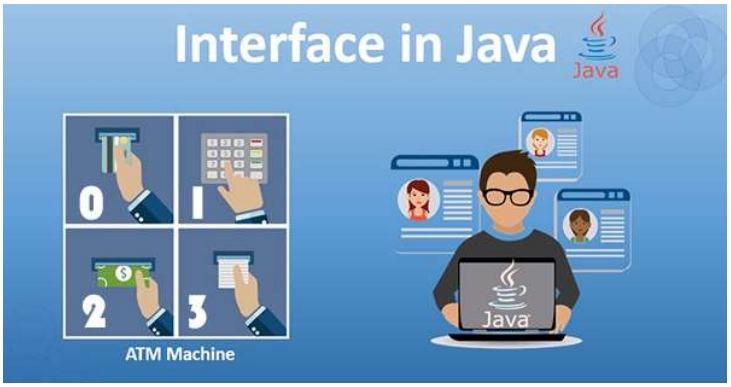
```
1.  boolean answer = listePoupeesRusse.stream()
2.       .map(e -> e.isFerme())
3.       . allMatch(n-> n);
```

## *Partie 2 : JavaFx (10pts)*

Soit la classe **Paiement** avec les attributs suivants : *code, libelle et type*Paiement.

*Cette classe admet :*

- les accesseurs aux différents attributs de la classe.
- Un **constructeur** permettant d'initialiser les attributs code et libelle d'un objet.




- la méthode **toString ( )** permettant d'afficher les informations de la paiement en cours.

Soit aussi : la classe ListePaiement qui contient une liste de quatre instances de paiement.

***Question 4.*** **Couche de présentation (View) (5=2.5+2.5pts)**

1. Dessiner le **graphe scène** de l'interface (figure précédente) qui comprend trois zones distinctes. Zone01 : le titre. Zone02 : 4 buttons et label. Zone03 : une image (logo.jpg).
2. Implémenter la méthode "**dessiner**" dans la classe "**InterfacePaiement** " qui va créer seulement la zone02 de l'interface graphique décrite ci-dessus. Cette méthode devra renvoyer une instance du composant ou du conteneur correspondant à cette interface.

***Question 5.*** **Couche de contrôle (5=1.5+1.5+2pts)**

1. Ecrire la méthode "**controlePaiement t**" dans la classe "Controleur". Cette méthode devra créer une instance de la vue (view), une instance du modèle (modèle) et une couche de gestion des actions. La fonction de cette méthode est d'assurer l'affichage de l'interface et de gérer les actions des boutons (Le clique sur le button d'indice i permet d'afficher l'instance de paiement de la ListePaiement d'indice i avec i 0 à 3).

**Annexe 01 :**

**Les fonctions de Stream sous forme d'expression lambda**

- Stream<T> filter(Predicate<? super T> predicate)
- void forEach(Consumer<? super T> action)
- boolean allMatch(Predicate<? super T> predicate)
- <R> Stream<R> map(Function<? super T, ? extends R> mapper)
- Stream<T> sorted(Comparator<? super T> comparator)

a. Le **BorderPane** qui vous permet de diviser une zone graphique en cinq parties : top, down, right, left et center.
b. La **Hbox** qui vous permet d'aligner horizontalement vos éléments graphiques.
c. La **VBox** qui vous permet d'aligner verticalement vos éléments graphiques.
d. Le **GridPane** permet de créer une grille d'éléments organisés en lignes et en colonnes

**Code source**

**Les instructions Javafx**

```
private HBox entete;

entete.getChildren().add(liste);
entete.getChildren().addAll(zoneDetail);
```

```
Label label = new Label("My Label");
```

```
tabButton[i][j] = new Button("" + i + " " + j);
label.setOnAction(event -> {
                    gestionAction(jj);
                });
```

```
bPane.setTop(new TextField("Top"));
bPane.setBottom(new TextField("Bottom"));
bPane.setLeft(new TextField("Left"));
bPane.setRight(new TextField("Right"));
bPane.setCenter(new TextField("Center"));
BorderPane bPane = new BorderPane();
```

```
zoneDetail.add(new Label("Nom"), 0, 0);
zoneDetail.add(new Label("score"), 0, 1);
private GridPane zoneDetail;
```

```
TextField b = new TextField();
b.getText()
```

```
Image img = new Image("logo.png");
ImageView view = new ImageView(img);
button.setGraphic(view);
```

***Bon Travail.***