TP5

Exercice 1

1)Dans un premier temps, on cherche à définir un Container. Un conteneur possède une destination sous forme de chaîne de caractères ainsi qu'un poids (weight en Anglais) qui est une valeur entière. Il ne doit pas être possible de créer un conteneur avec des valeurs invalides : la destination doit exister et le poids doit être positif ou nul. Écrire le type Container de telle façon à ce que le code suivant fonctionne :

```
public record Container(String destination, int weight) {

    public Container{
        Objects.requireNonNull(destination, "destinaion is null");
        if(weight<0) {
            throw new IllegalArgumentException("le poid doit être positif");
        }
    }
}
```

2)On veut maintenant introduire la notion de Manifest, un manifeste contient une liste de conteneurs. Pour l'instant, un manifeste définit une seule méthode add(conteneur) qui permet d'ajouter un conteneur au manifeste. Il ne doit pas être possible d'ajouter un conteneur null. Écrire le type Manifest tel que le code suivant fonctionne :

```
public class Manifest {
    private ArrayList<Container> containers;

    public Manifest() {
        containers = new ArrayList<>();
    }

    public void add(Container container) {
        Objects.requireNonNull(container, "container is null");
        containers.add(container);
    }
}
```

3) On souhaite maintenant pouvoir afficher un manifeste. Cela revient à afficher chaque conteneur sur une ligne, avec un numéro, 1 pour le premier conteneur, 2 pour le suivant, etc, suivi de la destination du conteneur ainsi que de son poids. Le formatage exact pour une ligne est :

-container

```
    @Override
    public String toString() {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append(destination)
            .append(" ")
            .append(weight)
```

```
            .append("kg");
        return stringBuilder.toString();
    }
```

-manifest

```
    @Override
    public String toString() {
        StringBuilder stringBuilder = new StringBuilder();
        int i=1;
        for(var container : containers) {
            stringBuilder.append(i)
              .append(". ")
              .append(container)
              .append("\n");
              i++;
        }
        return stringBuilder.toString();
    }
```

4) Un porte-conteneur, comme son nom ne l'indique pas, peut aussi transporter des passagers. Un Passenger est défini par une destination uniquement, les passagers ne sont pas assez lourds pour avoir un vrai poids. Dans un premier temps, définir un Passenger afin que l'on puisse créer un passager uniquement avec sa destination. Puis expliquer comment modifier Manifest pour que l'on puisse enregistrer aussi bien des conteneurs que des passagers. Pour l'affichage, un passager affiche la destination ainsi que "(passenger)" entre parenthèse (cf le code plus bas). Écrire le code de Passenger et modifier le code de Manifest de telle façon que le code ci-dessous fonctionne.

-Nous allons créer une interface OnCargo qui possedera tous les atribus de nos objets.

```
package fr.uge.manifest;

public interface OnCargo {
    public abstract int weight();
    public abstract String destination();
}
```

-La liste container est changé en liste de OnCargo. Grace au 2 methodes add on pourra ajouter à notre liste des conatiners ou des passenger. En fonction du type de l'objet qui est donnée, une méthode sera utilisé plutot qu'une autre.

```
package fr.uge.manifest;
import java.util.ArrayList;

public class Manifest {
    private ArrayList<OnCargo> onCargo;
    public Manifest() {
        onCargo = new ArrayList<>();
    }

    public void add(Container container) {
```

```
        onCargo.add(container);
    }

    public void add(Passenger passenger) {
        onCargo.add(passenger);
    }

    @Override
    public String toString() {
        StringBuilder stringBuilder = new StringBuilder();
        int i=1;
        for(var onCargo : onCargo) {
            stringBuilder.append(i)
              .append(". ")
              .append(onCargo)
              .append("\n");
              i++;
        }
        return stringBuilder.toString();
    }
}
```

5) On souhaite ajouter une méthode price à Manifest qui calcule le prix pour qu'un conteneur ou qu'un passager soit sur le bateau. Le prix pour un passager est 10. Le prix pour un conteneur est le poids du conteneur multiplié par 2. Ajouter une méthode price à Manifest et faites en sorte que le prix soit calculé correctement.

-Ajout de la methode prix

```
package fr.uge.manifest;

public interface OnCargo {
    public abstract int price();
    public abstract int weight();
    public abstract String destination();
}
```

-Override sur la methode prix() pour le record Cargo, elle va retourner le poid du cargo*2.

```
public record Container(String destination, int weight) implements OnCargo{

    public Container{
        Objects.requireNonNull(destination, "destinaion is null");
        if(weight<0) {
            throw new IllegalArgumentException("le poid doit être positif");
        }
    }

    @Override
    public int price() {
        return this.weight*2;
    }
```

-Override sur la methode prix() pour le record Passenger, elle va retourner 10.

```
public record Passenger(String destination) implements OnCargo{
    public Passenger{}

    @Override
    public int price() {
        return 10;
    }

    @Override
    public int weight() {
        return 10;
    }
```

6) On veut maintenant rajouter une méthode weight à Manifest qui renvoie le poids total en considérant qu'un passager n'a pas de poids.

```
    public int weight() {
        int total=0;
        for(var i : onCargo) {
            total=total+i.weight();
        }

        return total;
    }
```

7) Il arrive que l'on soit obligé de décharger tous les conteneurs liés à une destination s'il y a des problèmes d'embargo (quand un dictateur se dit qu'il s'offrirait bien une partie d'un pays voisin par exemple). Dans ce cas, il faut aussi supprimer tous les conteneurs liés à cette destination au niveau du manifeste (mais pas les passagers). Pour prendre en compte cela, on introduit une méthode removeAllContainersFrom(destination) qui supprime tous les conteneurs liés à une destination. S'il n'y a pas de conteneur pour cette destination, on ne fait rien. Modifier le code pour introduire cette méthode pour que l'exemple ci-dessous fonctionne :

-on va créer une methode removeAllContainerFrom qui utilisera un Iterator pour parcourir la ArrayListe onCargo. Si l'itérateur est égal à la destination et que sa methode isContainer renvoie true, l'élement sera supprimé.

```
    public void removeAllContainerFrom(String destination) {
        Iterator<OnCargo> iterator = onCargo.iterator();
            while(iterator.hasNext()) {
                OnCargo onCargo = iterator.next();
                if(onCargo.destination().equals(destination) && onCargo.isContainer())
{
                    iterator.remove();
                }
            }
    }
```

-ajout de la methode isContainer();

```java
public interface OnCargo {
    public abstract int price();
    public abstract int weight();
    public abstract String destination();
    public abstract boolean isContainer();
}
```

-methode isContainer dans le record Container

```java
    @Override
    public boolean isContainer() {
        return true;
    }
```

-methode isContainer dans le record Passenger

```java
    @Override
    public boolean isContainer() {
        return false;
    }
```

8) Pour résoudre la question précédente, au lieu de faire des appels de méthode, on peut aussi utiliser instanceof. Expliquer comment on peut utiliser instanceof. Puis expliquer, selon vous, quel est le problème d'utiliser instanceof dans ce contexte et pourquoi on ne doit pas l'utiliser.

-instanceof permet de tester le type d'un objet. On pourrais utiliser instanceof pour tester si l'objet est un container, et si oui, le supprimer si sa destination est celle indiqué. On pourrais utliliser instanceof pour résoudre la question précédante mais utiliser des appels de méthode est plus propre. Il est préférable de modifier l'interface, le code plus haut, pour une meilleur maintenance et une meilleur lisibilité.