TP1 - Rappel de notions de programmation objet

Max Ducoudré - INFO1

Exercice 2 - YMCA

1. Écrire le code de VillagePeople de telle façon que l'on puisse créer des VillagePeople avec leur nom et leur sorte. Par exemple,

```
    var lee = new VillagePeople("Lee", Kind.BIKER);
    System.out.println(lee);  // Lee (BIKER)
```

On fabrique un record `VillagePeople` avec deux paramètres car un Village People semble être non-mutable puis on impémente une méthode `toString()`

```
public record VillagePeople(String name, Kind kind) {
    public VillagePeople {
        Objects.requireNonNull(name);
        Objects.requireNonNull(kind);
        if(name.isEmpty()) throw new IllegalArgumentException("The name cannot
be empty");
    }
    @Override
    public String toString() {
        return name + " ("+kind+")";
    }
}
```

2. On veut maintenant introduire une maison House qui va contenir des VillagePeople. Une maison possède une méthode add qui permet d'ajouter un VillagePeople dans la maison (Il est possible d'ajouter plusieurs fois le même). L'affichage d'une maison doit renvoyer le texte "House with" suivi des noms des VillagePeople ajoutés à la maison, séparés par une virgule. Dans le cas où une maison est vide, le texte est "Empty House".

On fabrique une classe House

```
public final class House {

    //2. On initialise une liste `villages` people en private, ce qui ne
laissera pas la possibilité de la modifier en dehors de la classe
    private final List<VillagePeople> villages = new ArrayList<VillagePeople>
();

    //2. La méthode add permet d'ajouter un village people à la liste. La pré-
```

```
condition permet de vérifier que l'utilisateur entre un village valide
    public void add(VillagePeople village) {
        Objects.requireNonNull(village);
        villages.add(village);
    }

    // 3. La méthode toString utilise l'API des streams pour afficher les noms
des villages proprement
    @Override
    public String toString() {
        return villages.isEmpty()
                ? "Empty House"
                : "House with " +
                   villages.stream()
                    .map(VillagePeople::name)
                    .sorted()
                    .collect(Collectors.joining(", "));
    }
}
```

4. En fait, avoir une maison qui ne peut accepter que des VillagePeople n'est pas une bonne décision en termes de business, ils ne sont pas assez nombreux. YMCA décide donc qu'en plus des VillagePeople ses maisons permettent maintenant d'accueillir aussi des Minions, une autre population sans logement.
   On souhaite donc ajouter un type Minion (qui possède juste un nom name) et changer le code de House pour permettre d'ajouter des VillagePeople ou des Minion. Un Minion affiche son nom suivi entre parenthèse du texte "MINION".

Pour pouvoir manipuler les `Minion` et les `VillagePeople` de la même manière dans House, il vas falloir ajouter une interface qu'on vas appeller `Resident`

On fabrique l'interface `Resident`

```
 interface Resident permits {
    String name();
}
```

On fabrique le record `Minion`

```
public record Minion(String name) implements Resident{

    public Minion {
        Objects.requireNonNull(name);
        if(name.isEmpty()) throw new IllegalArgumentException("The name cannot
be empty");
    }
```

```
    @Override
    public String toString() {
        return name + " (MINION)";
    }
}
```

On implémente l'interface dans le record `VillagePeople`

```
public record VillagePeople(String name, Kind kind) implements Resident{ /* ...
*/ }
```

La classe `House` vas donc désormais manipuler une liste de `Resident`

```
public final class House {
    private final List<Resident> residents = new ArrayList<Resident>();

    public void add(Resident resident) {
        Objects.requireNonNull(resident);
        residents.add(resident);
    }

    @Override
    public String toString() {
        return residents.isEmpty()
                ? "Empty House"
                : "House with " +
                   residents.stream()
                    .map(Resident::name)
                    .sorted()
                    .collect(Collectors.joining(", "));
    }
}
```

5. On cherche à ajouter une méthode averagePrice à House qui renvoie le prix moyen pour une nuit sachant que le prix pour une nuit pour un VillagePeople est 100 et le prix pour une nuit pour un Minion est 1 (il vaut mieux être du bon côté du pistolet à prouts). Le prix moyen (renvoyé par averagePrice) est la moyenne des prix des VillagePeople et Minion présent dans la maison.
   Écrire la méthode averagePrice en utilisant le polymorphisme (late dispatch) pour trouver le prix de chaque VillagePeople ou Minion.

Pour calculer le prix moyen, on vas ajouter une méthode price() à l'interface `Resident`. Ce qui nous permettra de calculer le prix moyen des `Minion` et des `VillagePeople`

```
interface Resident {
    String name();
```

```
    int price();
}
```

On implémente la méthode `price()` dans les records `Minion` et `VillagePeople`

```
public record Minion(String name) implements Resident{
    /* ... */
    @Override
    public int price() {
        return 1;
    }
}
```

```
public record VillagePeople(String name, Kind kind) implements Resident{
    /* ... */
    @Override
    public int price() {
        return 100;
    }
}
```

On peux désormais implémenter la méthode `averagePrice()` dans la classe `House` avec l'API des streams

```
public final class House {
    /* ... */
    public double averagePrice() {
        return
villages.stream().mapToInt(Resident::price).average().orElse(Double.NaN);
    }
}
```

6. En fait, cette implantation n'est pas satisfaisante car elle ajoute une méthode publique dans VillagePeople et Minion alors que c'est un détail d'implantation. Au lieu d'utiliser la POO (programmation orienté objet), on va utiliser la POD (programmation orienté data) qui consiste à utiliser le pattern matching pour connaître le prix par nuit d'un VillagePeople ou un Minion. Modifier votre code pour introduire une méthode privée qui prend en paramètre un VillagePeople ou un Minion et renvoie son prix par nuit puis utilisez cette méthode pour calculer le prix moyen par nuit d'une maison.

Pour utiliser la POD, on vas directement ajouter une méthode `priceOf` dans la classe `House` pour avoir tous les prix au même endroit

```
public final class House {
    /* ... */
    private int priceOf(Resident resident) {
        return switch(resident) {
            case VillagePeople vp -> 100;
            case Minion m -> 1;
        };
    }
}
```

Il nous faut donc ensuite retirer la méthode `price()` de `Minion`, `VillagePeople` et `Resident`

La méthode averagePrice devient donc :

```
public double averagePrice() {
    return villages.stream().mapToInt(resident ->
getPrice(resident)).average().orElse(Double.NaN);
}
```

7. L'implantation précédente pose problème : il est possible d'ajouter une autre personne qu'un VillagePeople ou un Minion, mais celle-ci ne sera pas prise en compte par le pattern matching. Pour cela, on va interdire qu'une personne soit autre chose qu'un VillagePeople ou un Minion en scellant le super type commun.

Sachant que l'interface n'est manipulée que pour `Minion` et `VillagePeople`, on peux la sceller

```
sealed interface Resident permits Minion, VillagePeople {
    String name();
}
```

8. On veut périodiquement faire un geste commercial pour une maison envers une catégorie/sorte de VillagePeople en appliquant une réduction de 80% pour tous les VillagePeople ayant la même sorte (par exemple, pour tous les BIKERs). Pour cela, on se propose d'ajouter une méthode addDiscount qui prend une sorte en paramètre et offre un discount pour tous les VillagePeople de cette sorte. Si l'on appelle deux fois addDiscount avec la même sorte, le discount n'est appliqué qu'une fois.

Pour ce faire, on vas ajouter un champ `Map<Kind, Boolean> discounts` dans la classe `House` qui vas nous permettre de savoir si on a déjà appliqué un discount pour une sorte

```
public final class House {
    /* ... */
    private final Map<Kind, Boolean> discounts = new HashMap<Kind, Boolean>();

    /* ... */
```

```
    // Quand on ajoute un discout, on ajoute la sorte dans la map avec la
valeur true.
    public void addDiscount(Kind kind) {
        Objects.requireNonNull(kind);
        discounts.put(kind, true);
    }

    // méthode private pour récupérer le discount d'une sorte
    private bool isDiscounted(Kind kind) {
        Objects.requireNonNull(kind);
        return discounts.getOrDefault(kind, false);
    }

    // Il faut donc modifier la méthode priceOf pour appliquer le discount de
80 %si il y en a un
    private int priceOf(Resident resident) {
        return switch(resident) {
            case VillagePeople vp -> isDiscounted(vp.kind()) ? 20 : 100;
            case Minion m -> 1;
        };
    }
    /* ... */

}
```

9. Enfin, on souhaite pouvoir supprimer l'offre commerciale (discount) en ajoutant la méthode removeDiscount qui supprime le discount si celui-ci a été ajouté précédemment ou plante s'il n'y a pas de discount pour la sorte prise en paramètre.

La méthode removeDiscount vas donc supprimer la sorte de la map si elle existe

```
public class House {
    /* ... */
    public void removeDiscount(Kind kind) {
        Objects.requireNonNull(kind);
        if(!discounts.containsKey(kind)) throw new IllegalStateException("There
is no discount for this kind");
        discounts.remove(kind);
    }
    /* ... */
}
```

10. Faire en sorte que l'on puisse ajouter un discount suivi d'un pourcentage de réduction, c'est à dire un entier entre 0 et 100, en implantant une méthode addDiscount(kind, percent). Ajouter également une méthode priceByDiscount qui renvoie une table associative qui a un pourcentage renvoie la somme des prix par nuit auxquels on a appliqué ce pourcentage (la somme est aussi un entier). La somme totale doit être la même que la somme de tous les prix par nuit (donc ne m'oubliez pas les

minions). Comme précédemment, les pourcentages ne se cumulent pas si on appelle addDiscount plusieurs fois.

Pour appliquer un certain pourcentage, le type du champ `discounts` vas devenir `Map<Kind, Boolean>` en `Map<Kind, Integer>` pour connaître le pourcentage de réduction par sorte

```
public final class House {
    /* ... */
    private final Map<Kind, Integer> discounts = new HashMap<Kind, Integer>();
    /* ... */

    // La méthode addDiscount vas directement appliquer 0
    public void addDiscount(Kind kind) {
        Objects.requireNonNull(kind);
        discounts.put(kind, 0);
    }

    // La méthode isDiscounted vas retourner le pourcentage de réduction et
devenir discountOf
    private int discountOf(Kind kind) {
        Objects.requireNonNull(kind);
        return discounts.getOrDefault(kind, 0);
    }

    // On ajoute une méthode addDiscount qui prend un pourcentage en paramètre
    public void addDiscount(Kind kind, int percent) {
        Objects.requireNonNull(kind);
        if(percent < 0 || percent > 100) throw new
IllegalArgumentException("The percent must be between 0 and 100");
        discounts.put(kind, percent);
    }

    // On modifie la méthode priceOf pour appliquer le discount avec un
pourcentage
    private int priceOf(Resident resident) {
        return switch(resident) {
            case VillagePeople vp -> (100 - discountOf(vp.kind())) * 100 / 100;
            case Minion m -> 1;
        };
    }

    // On ajoute la méthode priceByDiscount qui vas retourner une table
associative avec le pourcentage et le prix total
    public Map<Integer, Integer> priceByDiscount() {
        var result = new HashMap<Integer, Integer>();

        // Ajout des VillagePeople avec discount
        discounts
            .forEach((k, v) ->
                result.put(v,
                    villages.stream()
```

```
                        .filter(resident -> resident instanceof VillagePeople)
                        .map(resident -> (VillagePeople) resident)
                        .filter(villagePeople ->
discountOf(villagePeople.kind()) == v)
                        .mapToInt(this::getPrice)
                        .sum()
        ));

        // Récupération des VillagePeople sans discount
        var list = villages.stream()
                .filter(resident -> resident instanceof VillagePeople)
                .map(resident -> (VillagePeople) resident)
                .filter(villagePeople ->
!discounts.containsKey(villagePeople.kind()))
                .collect(Collectors.toList());

        // Comptage des MINION dans la maison
        int minionCount = (int)villages.stream().filter(resident -> resident
instanceof Minion).count();

        // Ajout des VillagePeople sans discount et des MINION dans la valeur 0
        if(list.size() > 0 || minionCount > 0) {
            result.put(0, list.stream().mapToInt(resident ->
getPrice(resident)).sum() + minionCount);
        }

        return result;
    }
}
```