Programmation orientée objet

Série 5

Joel Cavat / 2020

Exercices

5.1 Exercice *(Properties)*

Réécrivez correctement une classe `Properties` qui supprime totalement la notion d'héritage. Choisissez judicieusement la structure à utiliser pour stocker des propriétés.

```
class Properties {

  public String getProperty(String key) {
    /* TODO */
  }
  public String getPropertyOrElse(String key, String defaultValue) {
    /* TODO */
  }
  public void addProperty(String key, String value) {
    /* TODO */
  }
  public List<String> keys() {
    /* TODO */
  }
  public List<String> values() {
    /* TODO */
  }
  public ????? allProperties() {
    /* TODO */
  }
}
```

5.2 Exercice *(Echecs)*

Vous devez réaliser le modèle de classes d'un jeu d'échecs. Pour cette première itération, nous allons nous concentrer sur un sous-ensemble de fonctionnalités de ce jeu.

Vous devez modéliser les concepts de pièces, d'échiquier et de cases. La mise en oeuvre des pièces spécifiques (reine, roi, fou, ...) se fera lors d'une prochaine itération.

Réalisez le code source minimal permettant de respecter les fonctionnalités demandées:

1. Un échiquier est composé de 64 cases.
2. Une case est caractérisée par une lettre et un numéro.
3. Une fonctionnalité permet de vérifier que le déplacement d'une pièce est légal. Le comportement est spécifique à chaque pièce. Ne modélisez pas les pièces spécifiques, juste une abstraction.
4. Il est possible de déplacer une pièce vers une case si le mouvement est légal. Le comportement est le même pour toutes les pièces. Si le mouvement est légal, la case se voit assigner la pièce.
5. Une pièce a une référence sur son échiquier.
6. Il est possible de déterminer si la pièce est toujours sur l'échiquier. Si tel est le cas, il est possible de connaître la case.

Corrigé

```
public class Chessboard {
  private Case[] cases = new Case[64]; // 1.
}
```

```
public class Case {
  // 2.
  private char col;
  private int row;
  private Piece p;
  public Case(char col, int row) {
    this.col = col;
    this.row = row;
  }
  public void assign(Piece p) {
    this.piece = p;
  }
}
```

```
public interface Piece {
  boolean isLegalMove(Case c); // 3.
  default void move(Case c) { // 4.
    if(this.isLegalMove(c)){
      c.assign(this);
    }
  }
```

```
  Chessboard chessboard(); // 5.
  default boolean isOnChessboard() { // 6.
    for(Case c: chessboard()) {
      if(c == this) {
        return true;
      }
    }
    return false;
  }
}
```

5.3 Exercice *(Transporteurs)*

Réalisez une application permettant de modéliser des transporteurs de marchandises. Un transporteur contient des conteneurs. Un conteneur renferme des articles. Il est possible de récupérer les conteneurs d'un transporteur un à un à l'aide d'une méthode takeNext(). Il est également possible de récupérer le prix d'un article. Nous pouvons donc déduire le prix total d'un conteneur en sommant le prix des articles qu'il renferme. Le modèle actuel propose un bateau comme moyen de transport et une voiture comme unique article. Complétez le code ci-dessous pour respecter les exigences demandées au paragraphe précédent.

```
interface Transport {
   /**
    * Prend un conteneur du transport
    * @return le prochain conteneur. Retourne une exception s'il n'y
    * a plus de conteneur
    */
    public Container takeNext();
}

interface Article {
   /**
    * Calcul le prix de l'article
    * @return le prix de l'article
    */
    public double price();
}

class Car implements Article {
    private String brand;
    private double price;
    public Car(String brand, double price) {
        this.brand = brand;
        this.price = price;
    }
    // A compléter

}
```

```
class Container {
    private Article[] articles;
    public Container(Article[] articles){
      // A compléter

    }
    public double totalPrice() {
      // A compléter

    }
}

class Boat implements Transport {
    private Container[] containers;
    private String name;
    public Boat(String name, Container[] containers) {

        // A compléter

    }
    // A compléter

}
```

5.4 Exercice *(StudentSimulator)*

Nous souhaitons simuler les conséquences d'une soirée d'intégration des étudiants en mesurant les effets des boissons sur leur socialisation. Lors de la première itération du développement, vous allez modéliser les bases d'un tel simulateur.

Une boisson indique un bénéfice social et un prix si on la consomme. Une bière a un bénéfice social de 3 et un prix de 4 alors qu'un soda retourne un bénéfice social de 1 pour un prix de 3.

Un être social peut prendre une boisson et on peut récupérer le rang social courant (L'addition des bénéfices sociaux de chaque boisson prise). Un étudiant et un enseignant sont tous deux des êtres sociaux.

Un étudiant possède un crédit initial de 20 et un rang social de zéro lors de sa création. Chaque fois qu'il prend une boisson, il faut vérifier s'il a assez d'argent. S'il a assez d'argent, il est nécessaire d'augmenter son rang social et de déduire le prix de la boisson. S'il n'a pas assez d'argent, il ne se passe rien.

Un enseignant n'a pas de crédit. Toutes les boissons lui sont généreusement offertes. Par contre, l'enseignant étant beaucoup moins drôle qu'un étudiant, son rang social n'est augmenté que de moitié à chaque consommation.

Complétez le code ci-dessous afin d'ajouter les concepts demandés dans le cahier des charges :

```
interface Drink {
    public int socialBenefit();
    public int price();
}

interface Socializer { /* Etre social */
    public void take(Drink drink);

    // A compléter
}

// A compléter
```

5.5 Exercice *(Statut)*

Vous devez réaliser une libraire permettant de représenter l'état d'une machine. Plutôt que d'avoir un état binaire, nous souhaitons avoir trois états:

- **On** : Signifie que la machine fonctionne,
- **Off** : Signifie que la machine est éteinte,
- **Err** : Signifie que la machine est allumée mais, qu'il y a un problème de fonctionnement. L'erreur est encapsulée dans un attribut de la classe `Err`.

Nous simulons l'enclenchement d'une action sur la machine par la méthode statique du type `Status`:

```
1 public static Status process() { ... }
```

Cette méthode retourne aléatoirement un `Status` qui est l'un des étas cités au-dessus (`On`, `Off` ou `Err`).

Vous devez étendre les fonctionnalités de `Status` et ses sous-classes pour déterminer le contexte. Vous ne pouvez utiliser `isInstance()` ou `instanceof` et vous n'avez pas de droit de caster des objets. Utilisez plutôt des méthodes `isOn`, `isOff` et `isError`.

Réalisez ensuite un algorithme qui affiche des informations dans le terminal selon le contexte:

- Si `process()` retourne un objet de type `On`, il faut afficher "L'appareil fonctionne",
- Si `process()` retourne un objet de type `Off`, il faut afficher "L'appareil est éteint",
- Si `process()` retourne un objet de type `Err`, il faut afficher "L'appareil est instable" suivi du message d'erreur.

Extrait d'utilisation:

```
Status s1 = Status.process();
if( s1.isOn() ) {
  System.out.println("L'appareil fonctionne");
} else( s1.isOff() ) {
  System.out.println("L'appareil est éteint");
} else {
  System.out.println("L'appareil est instable: " + s1.getErrorMessage());
}
Status s2 = Status.makeError("Oups");
```

5.6 Exercice *(Périphériques)*

Nous souhaitons gérer un parc de périphériques en nous focalisant sur sa maintenance. Il existe plusieurs types de périphériques :

- Ordinateurs (`Computer`)
- Imprimantes (`Printer`)
- Caméras de surveillance (`Camera`)
- Machine à cafés (`CoffeeMachine`)

Chaque périphérique possède un identifiant unique sous forme d'une chaine de caractères.Utilisez la classe utilitaire `UUID` pour générer un identifiant unique. Nous pouvons appliquer plusieurs méthodes sur un périphérique :

- `start()` : démarre (Wake-On-Line) un périphérique uniquement s'il est éteint
- `reboot()` : redémarre le périphérique s'il fonctionne ou s'il a un problème
- `shutdown()` : éteins le périphérique s'il n'est pas déjà éteint
- `status()` : retourne le statut du périphérique
- `id()` : retourne l'id unique du périphérique

Si un périphérique ne fonctionne pas correctement, supposez que les erreurs possibles soient :

- "Material Communication Error"
- "Updating Windows 3.1 Error"
- "Protocol Error"
- "Undefined Error"

Nous souhaitons enregistrer chaque périphérique dans une base de données (extrait de code à compléter plus bas). Nous souhaitons réaliser deux versions.

- la première version simulera une base de données à l'aide d'une `List`
- la deuxième version utilisera une `HashMap`

```
public interface Database {

  public void add(Device d); // ajoute ou remplace un périphérique

  /* retourne le périph qui a l'id spécif.
   * une exception sinon */
  Device device(String id);
  List<Device> devices();

  default boolean exists(String id) {

    /* Code à compléter */

  }
  // retourne le nombre de périphériques enregistrés dans la BD
  public default int count() {

    /* Code à compléter */
```

```
  }

  /* supprime le périph. qui a l'id spécif. et le retourne
   * une exception sinon */
  public Device remove(String id);
}
```

Dans votre programme principal, enregistrer les périphériques dans une base de données de manière automatique. Réalisez une fonctionnalité (`populate()`) qui prend une base de données et un nombre de périphériques à insérer dedans.

L'état initial du périphérique se fait selon une probabilité donnée ci-dessous :

Statut	Probabilité
On	50%
Off	30%
Err	20%

Puis, lors d'un démarrage ou redémarrage :

Statut	Probabilité
On	70%
Err	30%

Votre application doit pouvoir être testée avec plusieurs milliers de prériphériques.

Comptez le nombre de démarrages et de redémarrages nécessaires pour obtenir un parc stable (100% des périphériques fonctionnels). Pour ce faire, réalisez une fonctionnalité `stabilize()` qui prend une base de données de périphériques et retourne le nombre de démarrages et le nombre redémarrages. Imaginez une structure qui permet de retourner ces deux valeurs.

Enfin, réalisez une dernière fonctionnalité `partition()` qui prend également une base de données et retourne trois listes (périphériques `on`, `off` et `err`).

5.7 Exercice *(Héritage - Test 2019)*

Nous avons vu durant la théorie et les travaux pratiques les dangers de l'héritage et une solution pour y remédier. Dans le code ci-dessous, nous souhaitons que la classe `B` expose uniquement une méthode `test()` (et non les méthodes `a` et `b`). Réécrivez la classe `B` uniquement en supprimant cette relation d'héritage tout en conservant le comportement de `A` pour cette méthode.

```
class A {
    public A() {}
    public void test() {
        a();
        ...
        b();
    }
    public void a() {}
    public void b() {}
}
class B extends A {
    public B() {}
}
```

```
// Fichier B.java
class B { // extends supprimé


}
```

5.8 Exercice *(Test 2019)*

Vous devez réaliser une implémentation d'une liste dynamique d'entiers. Votre implémentation, appelée `ArrayListInt` doit **impérativement** utiliser un tableau statique d'entiers pour simuler le comportement d'une telle liste.

Voici une utilisation possible:

```java
public class Test2a {
    public static void main(String[] args) {

        ListInt list = new ArrayListInt();
        list.insert(0);
        list.insert(3);
        list.insertAll(2,1); // insertAll prend un nombre arbitraire d'éléments

        System.out.println( "Size: " + list.size() );

        for (int i = 0; i < list.size(); i+=1) {
            int v = list.get(i);
            System.out.println("Value: " + v);
        }

        list.clear();
        System.out.println( "Size: " + list.size() );

        /*
         * Cet exemple afficherait
         * Size: 4
         * Value: 0
         * Value: 3
         * Value: 2
         * Value: 1
         * Size: 0
         */
    }
}
```

Vous êtes libre d'utiliser:

- un tableau de type `int` primitif (`int[]`) ou
- un tableau d'objets `Integer` (`Integer[]`)

Contraintes supplémentaires

- Implémentez tous les composants pour que le code ci-dessus compile et s'exécute correctement
- Votre implémentation doit respecter le contrat de `ListInt`
- Les méthodes `isEmpty` et `addAll` doivent être **concrète** dans `ListInt`
- L'implémentation de `ArrayListInt` réserve initialement une taille de tableau de 10 éléments. A chaque dépassement de capacité, vous devez allouer 10 éléments supplémentaires.

- **Utilisez uniquement ces fonctionnalités** suivantes sur les tableaux statiques:
  - l'attribut `length` qui retourne la taille d'un tableau
  - la notation crochet pour modifier ou extraire une valeur du tableau
  - la boucle de parcours
- Ecrivez lisiblement