

Programmation Orientée Objets avec Java

Chapitre 3 : Héritage et Polymorphisme

Stéphane Malandain / Yassin Rekik
Semestre d'automne 2023

Exercice 1

Soit les déclarations suivantes :

```
1 public class A {  
2     public void f(int i) { System.out.println("A: f(int)"); }  
3     public void f(short s) { System.out.println("A: f(short)"); }  
4 }
```

```
1 public class B extends A {  
2     public void f(short s) { System.out.println("B: f(short)"); }  
3 }
```

```
1 public class C extends B {  
2     public void f(int i) { System.out.println("C: f(int)"); }  
3     public void f(byte b) { System.out.println("C: f(byte)"); }  
4 }
```

```
1 A ab = new B();  
2 B bc = new C();  
3 A ac = new C();  
4 int i;  
5 byte b;  
6 short s;
```

Que vont afficher les appels ci-dessous. Précisez si l'appel provoque une erreur de compilation:

```
1 ab.f(i);  
2 bc.f(i);  
3 ac.f(i);  
4 ab.f(s);  
5 bc.f(s);  
6 ac.f(s);  
7 ab.f(b);  
8 bc.f(b);  
9 ac.f(b);
```

Correction :

```
jshell> ab.f(i);  
...> bc.f(i);  
...> ac.f(i);  
...> ab.f(s);  
...> bc.f(s);  
...> ac.f(s);  
...> ab.f(b);  
...> bc.f(b);  
...> ac.f(b);  
A : f(int)  
C : f(int)  
C : f(int)  
B : f(short)  
B : f(short)  
B : f(short)  
B : f(short)  
B : f(short)  
B : f(short)
```

Exercice 2

Soient les déclarations suivantes:

```
1 class A {  
2     public void f(int i) { ... }  
3     public void f(short s) { ... }  
4     public Serializable g(Integer i) { ... }  
5 }
```

Précisez pour chaque méthode ci-dessous s'il s'agit d'une surcharge ou d'une redéfinition.

```
1 class B extends A {  
2     public void f(short s) { ... }  
3     public void f(byte b) { ... }  
4     public void f(int i) { ... }  
5 }
```

2	redéfinition
3	surcharge
4	redéfinition

Précisez également pour chaque méthode ci-dessous si elle est valide et s'il s'agit d'une surcharge ou d'une redéfinition.

```

1 class C extends A {
2     public String g(Integer i) { ... }
3     public Serializable g(Integer i) { ... }
4     public Serializable g(Object o) { ... }
5 }

```

- 2 redéfinition
- 3 redéfinition
- 4 surcharge

Exercice 3

Soient les déclarations suivantes:

```

1 class A {}
2 class B extends A {}
3 class C extends A {}
4 class E extends A {}
5 class Z {
6     public void f(B b) { System.out.println("B"); }
7     public void f(C c) { System.out.println("C"); }
8     public void f(E e) { System.out.println("E"); }
9 }

```

Pour chaque ligne, précisez ce qui sera affiché ou le type d'erreur (exécution ou compilation):

```

1 Z z = new Z();
2 z.f( new A() );
3 z.f( new B() );
4 z.f( new C() );
5 z.f( new D() );
6 z.f( "COUCOU" );

```

```

1 class A {}
2 class B extends A {}
3 class C extends A {}
4 class E extends A {}
5 class Z {
6     public void f(B b) {System.out.println(x: "B");}
7     public void f(C c) {System.out.println(x: "C");}
8     public void f(E e) {System.out.println(x: "E");}
9 }
10
11 public class S3E3 {
12     public static void main(String[] args) {
13
14         Z z = new Z();
15         // z.f( new A() );
16         z.f( new B() ); // Affiche B
17         z.f( new C() ); // Affiche C
18         // z.f( new D() );
19         // z.f( "COUCOU" );
20     }
21 }

```

Exercice 4 : Jeu d'échec

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 : « Pièces », « Échiquier » et « 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.

```
1 public class Chessboard {
2     private Case[] cases = new Case[64]; // 1.
3 }
4 public class Case {
5     // 2.
6     private char col;
7     private int row;
8     public Case(char col, int row) {
9         this.col = col;
10        this.row = row;
11    }
12 }
13 public abstract class Piece {
14     protected Case c;
15     public Piece(Case c) { this.c = c; } // 3.
16     public abstract boolean isLegalMove(Case c); // 4.
17     public void move(Case c) { // 5.
18         if(this.isLegalMove(c)){
19             this.c = c;
20         }
21     }
22 }
```

Deuxième partie :

Maintenant que votre classe « Pièce » est définie. Proposer trois classes concrètes permettant de concrétiser cette classe. Il s'agit des classes : Fou, Tour et Cheval.

Exercice 5 : Simulateur Social

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 :

```
4 interface Drink {
5     public int socialBenefit();
6     public int price();
7 }
8
9 interface Socializer { /* Etre social */
10    public void take(Drink drink);
11    // A completer
12 }
13
14 // A completer

1 interface Drink {
2     public int socialBenefit();
3     public int price();
4 }
5 class Beer implements Drink {
6     public int socialBenefit() { return 3; }
7     public int price() { return 4; }
8 }
9 class Soda implements Drink {
10    public int socialBenefit() { return 1; }
11    public int price() { return 3; }
12 }
13
14
15 interface Socializer { /* Etre social */
16    public void take(Drink drink);
17    public int socialRank();
18 }
19
20 class Teacher implements Socializer {
21    private int socialRank;
22    public void take(Drink drink){
23        this.socialRank += drink.socialBenefit() / 2;
24    }
25    public int socialRank() { return this.socialRank; }
26 }
27 class Student implements Socializer {
28    private int socialRank;
29    private int money = 20;
30    public void take(Drink drink){
31        if(this.money >= drink.price()){
32            this.socialRank += drink.socialBenefit();
33            this.money -= drink.price();
34        }
35    }
36    public int socialRank() { return this.socialRank; }
37 }
38
```

Exercice 6 : Simulateur de Machine

Vous devez réaliser une librairie 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:

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

Cette méthode retourne aléatoirement un Status qui est l'un des états 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:

```
2 Status s1 = Status.process();
3
4 if( s1.isOn() ) {
5     System.out.println("L'appareil fonctionne");} else( s1.isOff() ) {
6     System.out.println("L'appareil est éteint");
7 } else {
8     System.out.println("L'appareil est instable: " + s1.getErrorMessage());
9 }
10 Status s2 = Status.makeError("Oups");
11
```

```

1  import java.util.Random;
2
3  interface Status {
4
5      static Status process() {
6          int chance = new Random().nextInt(bound: 3);
7          if( chance == 0 ) {
8              return new On();
9          } else {
10             return chance == 1 ? new Off() : new Err(msg: "Oops");
11          }
12      }
13
14      static Err makeError(String message) {
15          return new Err(message);
16      }
17
18      default boolean isOn() { return false; }
19      default boolean isOff() { return false; }
20      default String getErrorMessage() { throw new UnsupportedOperationException(message: "..."); }
21  }
22
23  class On implements Status {
24
25      public boolean isOn() { return true; }
26  }
27
28  class Off implements Status {
29      public boolean isOff() { return true; }
30  }
31
32
33  class Err implements Status {
34      private String msg;
35      public Err(String msg) {
36          this.msg = msg;
37      }
38      public String getErrorMessage() { return this.msg; }
39  }
40

```

```

41
42  public class StatusApp {
43      Run | Debug
44      public static void main(String[] args) {
45
46          Status s1 = Status.process();
47          if( s1.isOn() ) {
48              System.out.println(x: "L'appareil fonctionne");
49          } else if ( s1.isOff() ) {
50              System.out.println(x: "L'appareil est éteint");
51          } else {
52              System.out.println("L'appareil est instable: " + s1.getErrorMessage());
53          }
54          Status s2 = Status.makeError(message: "Oops");
55      }
56

```