Consolidation 2006-2007 - Durée : 2h

IB2 : Informatique de Base 2

Première année

Nancy-Université — Université Henri Poincaré

Tous documents interdits.

★ **Exercice 1.** Questions de cours. (4 pt)

▷ **Question 1.** Expliquez la différence entre *surcharger* une méthode et *rédéfinir* une méthode. Donnez des exemples. (1 pt)

▷ **Question 2.** Expliquez la différence entre une *classe abstraite* et une *interface*. Indiquer l'intérêt de chacun de ces deux concepts. (1 pt)

▷ **Question 3.** Explicitez les caractéristiques d'un algorithme récursif avec retour arrière (*back-tracking*). Vous illustrerez vos explications d'un exemple d'application d'un tel algorithme. (2 pt)

★ **Exercice 2.** Typage statique et typage dynamique. (4 pt)

On considère le code suivant :

```
1  package consoib2;
2
3  class A {
4      protected int x;
5
6      public A(int x) {
7          this.x = x ;
8      }
9      public void f(A a) {
10         System.out.println("f A de A");
11         x = x + a.x ;
12     }
13     public void f(B b) {
14         System.out.println("f B de A");
15         x = b.x;
16     }
17     public String toString() {
18         return "A, x = "+x;
19     }
20 }
21
22 class B extends A {
23     protected A a;
24     protected int x=1;
25
26     public B(A a, int x) {
27         super(x);
28         x = 2;
29         this.a = a ;
30     }
31
32     public void f(A a) {
33         System.out.println("f A de B");
34         this.a.x = a.x + x; 5
35         x = super.x + a.x;
36     }          4      5
37     public void f(B b) {
38         System.out.println("f B de B");
39         a.x = 1+b.x;
40         x = 3 + b.x ;
41     }
42     public String toString() {
43         return "B, x = "+x+", sx = "+super.x+", a = <"+a.toString()+">";
44     }
45 }
46
47
48 public class OnTeste {
49     public static void main(String[] args) {
50         A a = new A(3); B b = new B(a, 4);
51         System.out.println(b);
52         a.f(a);
```

```
53          System.out.println(b); //schema1
54
55          a = new A(3); b = new B(a, 4);
56          a.f(b);
57          System.out.println(b); //schema2
58
59          a = new A(3);b = new B(a, 4);
60          b.f(a);
61          System.out.println(b); //schema3
62
63          a = new A(3); b = new B(a, 4);
64          b.f(b);
65          System.out.println(b); //schema4
66      }
67  }
```

▷ **Question 1.** Une fois le programme compilé, on lance la commande : `java consoib2.OnTeste`. En vous aidant de schémas mémoire (4 schémas) dire ce qui s'affiche et expliquer pourquoi. (4 pt)

★ **Exercice 3.** Algorithme récursif "mystère". (3 pt)

On considère l'algorithme suivant :

```
1  package consoib2;
2
3  public class Mystere {
4      public String mystification(int i) {
5          if (i == 0)
6              return "0";
7          else if (i == 1)
8              return "1";
9          else {
10             if (i % 2 == 0)
11                 return mystification(i/2) + "0";
12             else
13                 return mystification(i/2) + "1";
14         }
15     }
16 }
```

On suppose que la valeur intialement donnée en paramètre à la fonction `mystere(int i)` est une valeur entière positive.

▷ **Question 1.** Montrer la terminaison de cet algorithme. (1 pt)

▷ **Question 2.** Explicitez l'appel `(new Mystere()).mystification(13)` en indiquant le détail de chaque appel récursif. (1 pt)

▷ **Question 3.** Que calcul cet algorithme ? (1 pt)

★ **Exercice 4.** Gestion des exceptions. (4 pt)

On considère l'interface `Stack` et la classe `Value` suivantes :

```
1  package consoib2;
2
3  class Value {
4      private String name;
5      private int value;
6
7      public Value(String n, int v) {
8          this.name = n;
9          this.value = v;
10     }
11     public String toString() {
12         return "<"+this.name+";"+this.value+">";
13     }
14 }
15
16 public interface Stack {
17
18     public boolean empty() ;
19
20     public void push(Value v) ;
```

```
21
22      public Value pop() throws EmptyStackException;
23
24      public Value peek() throws EmptyStackException;
25  }
```

- La classe `Value` décrit un couple <nom de la valeur, valeur entière stockée>,
- L'interface `Stack` décrit l'interface standard d'une pile de valeurs :
  - la méthode `boolean empty()` indique si la pile est vide ou non,
  - la méthode `void push(Value v)` empile une nouvelle valeur,
  - la méthode `Value pop()` retourne et dépile une valeur empilée,
  - la méthode `Value peek()` retourne une valeur empilée, sans la dépiler.
  - Les méthodes `Value pop()` et `Value peek()` ne peuvent pas retourner de valeur quand elles sont appelées sur une pile vide. Dans ce cas, une exception `EmptyStackException` doit être levée.

▷ **Question 1.** Écrire la classe `EmptyStackException` héritant de la classe `Exception` ? (1 pt)

▷ **Question 2.** Écrire une classe `LIFOStack` implémentant l'interface `Stack` qui réalise une pile du type *Last In, First Out* (la dernière valeur empilée est la première valeur à être dépilée). Pensez à lever une exception quand c'est nécessaire. (2 pt)

Indication : Comme structure interne de votre pile, vous utiliserez au choix un objet de classe `Vector`, `ArrayList` ou un tableau d'objets de type `Value` dont la taille sera dynamique.

▷ **Question 3.** Écrire une classe `TestStack` qui crée une instance de la classe `LIFOStack`, qui empile une valeur et essaye de la dépiler. N'oubliez pas de capturer les exceptions qui peuvent être levées. (1 pt)

★ **Exercice 5.** *"More is good... all is better"* (5 pt)

Le brave marchand du nom d'Ali Baba vient de rentrer dans la caverne des 40 voleurs et souhaite leur voler une partie de leur trésor. Malheureusement, Ali Baba ne peut porter qu'un coffre d'un poids limité. Il doit donc sélectionner intelligemment les objets qu'il souhaite prendre afin de maximiser son profit.

L'objectif de cet exercice est d'écrire un algorithme récursif permettant de résoudre ce problème. Chaque objet du trésor à un poids et une valeur qui lui est propre, un tel objet est décrit par la classe `Bijou` dont la définition est la suivante :

```
1  package consoib2;
2
3  public class Bijou {
4      private static int count = 0;
5      private int num;
6      private int poids;
7      private int valeur;
8
9      public Bijou(int p, int v) {
10         this.poids = p;
11         this.valeur = v;
12         this.num = count++;
13     }
14
15     public int getValeur() {
16         return this.valeur;
17     }
18     public int getPoids() {
19         return this.poids;
20     }
21     public String toString() {
22         return "Bijou "+num+" [valeur: "+this.valeur+" poids: "+this.poids+"]";
23     }
24 }
```

Les différents bijoux (dont on suppose la quantité illimitée) présents dans la caverne seront stockés dans un tableau dénommé `tresor`. Le contenu de ce tableau sera passé en paramètre au constructeur de la classe `RemplirCoffre`. Lors de l'exploration, la meilleure des solutions sera conservée dans un tableau nommé `meilleurButin`, une variable entière `valeurMeilleurButin` conservera la valeur de la meilleure solution (et donc du meilleur butin) trouvée jusqu'ici.

```
1    Bijou[] tresor;
2    int[] meilleurButin;
3    int valeurMeilleurButin = -1;
```

▷ **Question 1.** Il vous est demandé d'écrire le constructeur `public RemplirCoffre(int[] tresorCaverne)` de la classe `RemplirCoffre`

▷ **Question 2.** Implémenter la méthode `void afficheButin()` qui affiche si Ali Baba a trouvé une solution et dans ce cas affiche le nombre d'objets pris, leur poids, leur valeur et la valeur et le poids total du coffre.

▷ **Question 3.** Implémenter la méthode `int poids(int butin[], int niveauMaximum)` qui calcule le poids total de la solution `butin`. Le paramètre `niveauMaximum` limite le niveau d'exploration (on ne compte pas le poids des objets qui sont utilisés dans `butin` et dont l'indice est supérieur à `niveauMaximum`).

▷ **Question 4.** Implémenter la méthode `int valeur(int butin[], int niveauMaximum)` qui calcule la valeur totale de la solution `butin`. Le paramètre `niveauMaximum` limite le niveau d'exploration (on ne compte pas la valeur des objets qui sont utilisés dans `butin` et dont l'indice est supérieur à `niveauMaximum`).

▷ **Question 5.** Implémenter la méthode `void stockeSiMeilleurButin(int butin[], int niveauMaximum)` vérifie si la solution `butin` a une valeur plus élevée que la meilleure solution actuelle conservé dans le tableau `meilleurButin`. Si c'est le cas, la solution prise est recopiée en tant que meilleure solution. Il faut noter que cette méthode met également à jour la valeur de `valeurMeilleurButin` si nécessaire.

▷ **Question 6.** Implémenter la méthode `void cherche(int poidsCherche, int niveau, int butin[])` qui recherche la meilleure solution. Le paramètre `poidsCherche` contient le poids maximal que le coffre peut peser. Le paramètre `niveau` contient le niveau d'exploration (l'indice actuel dans le tableau `butin`). Le tableau `butin` contient la solution en cours d'élaboration.

▷ **Question 7.** Implémenter la méthode `void voler(int poids)` qui initialise le tableau `meilleurButin`, la valeur `valeurMeilleurButin` et effectue le premier appel à la méthode `cherche(int poidsCherche, int niveau, int butin[])`

▷ **Question 8.** Écrire une méthode `static void main(String args[])` résolvant le problème avec les paramètre suivants :

- La caverne contient les types de bijoux suivants :
  - Bijoux de poids 3 et de valeur 5,
  - Bijoux de poids 5 et de valeur 10,
  - Bijoux de poids 7 et de valeur 25,
  - Bijoux de poids 11 et de valeur 50,
  - Bijoux de poids 23 et de valeur 100,
- Le coffre peut contenir un poids maximal de 37 kg.