Exercices sur les interfaces

Fabrice Rossi

18 octobre 1999

1 Le type `Object`

1.1 Manipulations élémentaires

Exercice 1.1 :

Indiquer l'affichage produit par le programme suivant :

```
public class UpCast1 {
  public static void main(String[] args) {
    String s="test";
    Object o=s;
    System.out.println(o);
    o=o+" et test2"; // cas très particulier
    System.out.println(o);
    System.out.println(s);
    // impossible : char t=o.charAt(2);
    String u=(String)o;
    System.out.println(u);
  }
}
```

Expliquer pourquoi la ligne marquée *impossible* n'est pas correcte (ne compile pas).

Exercice 1.2 :

Indiquer l'affichage produit par le programme suivant :

```
public class UpCast2 {
  public static void main(String[] args) {
    int[] x={1,2};
    Object o=x;
    System.out.println(o);
    int[] y=(int[])o;
    y[0]=3;
    System.out.println(x[0]);
    // impossible : o[1]=6;
    String u=o.toString();
    System.out.println(u);
```

```
    String t=(String)o ;
  }
}
```

Expliquer pourquoi la ligne marquée *impossible* n'est pas correcte (ne compile pas).

Exercice 1.3 :

Indiquer l'affichage produit par le programme suivant :

```
public class UpCast3 {
  public static void main(String[] arg) {
    Object[] to=new Object[4] ;
    System.out.println(to[0]) ;
    to[0]="bla" ;
    to[1]=new int[] {1,2,3} ;
    to[2]=new StringBuffer("bli") ;
    to[3]=new double[] {0.5,-2.4} ;
    for(int i=0 ;i<to.length ;i++)
      System.out.println(to[i]) ;
    to[2]=to[0] ;
    System.out.println(to[2]) ;
  }
}
```

Exercice 1.4 :

Indiquer l'affichage produit par le programme suivant :

```
public class UpCast4 {
  public static Object test(Object o) {
    return o.toString() ;
  }
  public static void main(String[] arg) {
    String a="toto" ;
    System.out.println(test(a)) ;
    String b ;
    // impossible : b=test(a) ;
    b=(String)test(a) ;
    System.out.println(b) ;
    System.out.println(a.equals(b)) ;
    System.out.println(a==b) ;
    int[] t=(int[])test(a) ;
  }
}
```

Expliquer pourquoi la ligne marquée *impossible* n'est pas correcte (ne compile pas).

1.2 Algorithmique

Exercice 1.5 :

Ecrire une méthode qui à un tableau de type `Object[]`, `t`, associe un `String[]` contenant les références de tous les objets de type `String` "contenus" dans `t`.

Exercice 1.6 :

Ajouter à la classe `DTabObject` une méthode `void keepOnly(Object t)` qui supprime du `DTabObject` appelant toutes les références vers des objets dont le type est incompatible avec le type de l'objet désigné par `t`.

Exercice 1.7 :

Ecrire une méthode qui à un tableau de type `Object[]`, `t`, et une chaîne de caractères, `s`, associe l'indice du premier élément dont la représentation en chaîne de caractères (obtenue par `toString`) contient la chaîne `s`.

Exercice 1.8 :

Ecrire une méthode qui à un tableau de type `Object[]` associe un autre tableau du même type dans lequel aucun objet n'apparaît en double (aucune référence identique). Ecrire une méthode similaire qui compare les objets en utilisant `equals`.

Exercice 1.9 :

Compléter la classe `DTabObject` pour qu'elle propose les mêmes méthodes que la classe `ArrayList`.

Exercice 1.10 :

Ecrire une méthode qui à un tableau de type `Object[]` associe un tableau de type `Object[][]` contenant les mêmes références que le tableau d'origine, mais réparties par type : chaque ligne du tableau résultat contient exclusivement des références correspondant à des objets du même type.

2 Enveloppes pour les types fondamentaux

2.1 Analyse

Exercice 2.1 :

Donner l'affichage produit par le programme suivant. Pour chaque ligne en commentaires marquée "problème" indiquer la nature du problème et le moment de la détection (compilation ou exécution) :

```
public class Wrapper1 {
  public static void main(String[] arg) {
    StringBuffer sb=new StringBuffer();
    Character c=new Character('u');
    Character d=new Character('v');
    sb.append(c);
    sb.append(d);
    System.out.println(sb);
```

```
    /* problème : String s=(String)c; */
    Double y=new Double(3);
    /* problème : double x=y; */
    y=new Double(2.5*y.doubleValue());
    System.out.println(y);
    Object[] t={y,new Integer(3),new Boolean(true)};
    for(int i=0;i<t.length;i++)
      System.out.println(t[i]);
    /* problème : y=(Double)t[1]; */
    /* problème : y=3.5*y; */
  }
}
```

Exercice 2.2 :

Donner l'affichage produit par le programme suivant :

```
 1 : public class Wrapper2 {
 2 :   public static void main(String[] arg) {
 3 :     Class c=Double.TYPE;
 4 :     System.out.println(c);
 5 :     Double d=new Double(2.5);
 6 :     System.out.println(c.isInstance(d));
 7 :     Double e=new Double(2.5);
 8 :     System.out.println(e==d);
 9 :     System.out.println(e.equals(d));
10 :     Class u=e.getClass();
11 :     System.out.println(u.getName());
12 :     System.out.println(u.isInstance(d));
13 :     System.out.println(u.isInstance(e));
14 :     Class v=d.getClass();
15 :     System.out.println(u==v);
16 :     System.out.println(u.equals(v));
17 :     System.out.println(u.equals(c));
18 :   }
19 : }
```

Il se trouve que la ligne 15 affiche `true`. Que peut-on en déduire ?

Exercice 2.3 :

On considère les variables suivantes :

```
Object o;
Double d;
Number n;
Boolean b;
Integer i;
String s;
Character c;
```

Pour chaque ligne suivante, indiquez si la dernière affectation compile et, le cas échéant, si elle s'exécute sans erreur (les lignes sont considérées de façon indépendante les unes des autres) :

```
o=new Integer(2) ; d=(Double) o ;
n=new Double(2.5) ; o=(Object) n ;
b=new Boolean(true) ; n=b ;
i=new Number(2) ;
s="toto" ; c=s.charAt(0) ;
o="titi" ; s=(StringBuffer)o ;
c=new Character('u') ; o=c.toString() ; s=(String)o ;
d=new Double(3.2) ; i=(Integer)d ;
o=new Float(-3.9f) ; d=new Double(((Number)o).doubleValue()) ;
b=new Boolean(false) ; s=(String)b ;
```

2.2 Algorithmique

Exercice 2.4 :

Ecrire une méthode qui à un tableau de type `Object[]` associe la moyenne des valeurs numériques contenues dans ce tableau sous forme de références vers des objets de classes enveloppes numériques. On calculera la moyenne en `double`.

Exercice 2.5 :

Ecrire une méthode qui réalise le tri (sur place) d'un tableau de type `Number[]`. On pourra utiliser l'algorithme du tri par sélection (on choisit l'élément le plus petit qu'on place en première position, puis on trouve le plus petit élément restant, etc.). Expliquer pourquoi le tri ne sera pas nécessairement parfait (on considérera le cas d'une grande valeur de type `long`).

Exercice 2.6 :

Ecrire une méthode `produit` qui à deux `Number` associe leur produit sous forme d'un objet de type enveloppe numérique. On appliquera l'algorithme utilisé par `Java` pour les types fondamentaux (promotion vers le type le plus général et résultat de ce type).

Exercice 2.7 :

Ecrire une méthode qui à une chaîne de caractères, sous forme `String` ou `StringBuffer`, associe un tableau de type `Object[]` contenant tout les caractères de la chaîne, représentés par des objets `Character`. Ecrire une méthode qui réalise la transformation inverse, c'est-à-dire associe à un tableau `Object[]` la chaîne (`String`) constituée des caractères "contenus" dans le tableau.

Exercice 2.8 :

Ecrire une méthode qui à un tableau de `Boolean` (la classe, pas le type fondamental !) associe un `Boolean` conjonction de tous les éléments du tableau (i.e., `true` si et seulement si tous les éléments du tableau valent `true`).

3 Interfaces

3.1 Analyse

Exercice 3.1 :

On considère l'interface suivante :

```
public interface Bidule {
  public double map(double x) ;
  public int combine(int x,int y) ;
  public void put(String s) ;
}
```

Pour chacune des classes suivantes, indiquer si la classe implante correctement l'interface, en précisant le cas échéant la ou les source(s) d'erreur(s).

1. `BiduleImp1` :

```
public class BiduleImp1 implements Bidule {
  private String save ;
  public double map(double x) {
    return 2*x ;
  }
  public int combine(int x,int y) {
    return x+y ;
  }
  public void set(String s) {
    save=s ;
  }
}
```

2. `BiduleImp2` :

```
public class BiduleImp2 implements Bidule {
  private String save ;
  public double map(double x) {
    return Math.sqrt(x) ;
  }
  public int combine(int x,int y) {
    return x/y ;
  }
  public void put(String chaine) {
    save=chaine ;
  }
  public String get() {
    return save ;
  }
}
```

3. `BiduleImp3` :

```
public class BiduleImp3 implements Bidule {
  public double map(int x) {
    return x/2 ;
  }
  public double combine(int x,int y) {
    return ((double)x)/y ;
  }
  public void put(String s) {
  }
}
```

4. `BiduleImp4` :

```
public class BiduleImp4 implements Bidule {
  public double map(double x,double y) {
    return x+y ;
  }
  public int combine(int x,int y) {
    return x-y ;
  }
  public void put(String s) {
    System.out.println(s) ;
  }
}
```

Exercice 3.2 :

On considère les trois interfaces suivantes :

```
public interface InterA {
  public double f(double x) ;
}

public interface InterB extends InterA {
  public int g(int x) ;
}

public interface InterC extends InterA {
  public String h(String s) ;
}
```

Soit maintenant la classe `ImplABC` suivante :

```
public class ImplABC implements InterA,InterB,InterC {
  public double f(double x) {
    return 2*x ;
  }
  public int g(int x) {
    return x/2 ;
```

```
  }
  public String h(String s) {
    return "'"+s+"'";
  }
}
```

On considère le programme de test suivant :

```
 1 :  public class TestImplABC {
 2 :    public static void main(String[] arg) {
 3 :      ImplABC obj=new ImplABC();
 4 :      System.out.println(obj.f(-1.5));
 5 :      System.out.println(obj.g(5));
 6 :      System.out.println(obj.h("gnark"));
 7 :      InterA a=obj;
 8 :      System.out.println(a.f(2.34));
 9 :      System.out.println(a.g(-5));
10 :      System.out.println(a.h("pouic"));
11 :      InterB b=obj;
12 :      System.out.println(b.f(-4.37));
13 :      System.out.println(b.g(31));
14 :      System.out.println(b.h("titi"));
15 :      InterC c=obj;
16 :      System.out.println(c.f(13));
17 :      System.out.println(c.g(13));
18 :      System.out.println(c.h("toto"));
19 :      b=c;
20 :      a=c;
21 :      c=a;
22 :      c=b;
23 :    }
24 :  }
```

Indiquer les lignes qui sont refusées à la compilation (en précisant pourquoi). En supposant que ces lignes sont supprimées, indiquer l'affichage produit par le programme.

3.2 Abstraction

Exercice 3.3 :

L'algorithme de la section dorée permet de trouver un minimum local d'une fonction de $\mathbb{R}$ dans $\mathbb{R}$. En voici une description :

Données :

- un intervalle $[a, b]$ de $\mathbb{R}$;
- un point $x \in]a, b[$;
- une fonction f continue de $[a, b]$ dans $\mathbb{R}$ et telle que $f(a) > f(x)$ et $f(b) > f(x)$;

– un réel $\epsilon > 0$ représentant la précision de résolution.

Résultats : un intervalle $[u, v]$ de $\mathbb{R}$ et un point $y \in]u, v[$ tels que :

– $(v - u) < \epsilon \frac{v+u}{2}$;

– $f(u) > f(y)$ et $f(v) > f(y)$

1. on pose `u`=a, `v`=b et `y`=x
2. on pose `g`=$\frac{\sqrt{5}-1}{2}$
3. tant que `v-u`$\geq \epsilon$`(v+u)/2` :
   (a) si l'intervalle `]u,y[` est plus grand que l'intervalle `]y,v[` :
      i. placer dans `w` le résultat de `u+g*(y-u)`
   (b) sinon
      i. placer dans `w` le résultat de `v-g*(v-y)`
   (c) si f(`w`) > f(`y`)
      i. si `w<y`, placer `w` dans `u`
      ii. sinon, placer `w` dans `v`
   (d) sinon (on fait l'hypothèse simplificatrice que f(`w`) $\neq f$(`y`)) :
      i. si `w<y`, placer `y` dans `v` et `w` dans `y`
      ii. sinon, placer `y` dans `u` et `w` dans `y`
4. Résultat : les contenus des variables `u`, `v` et `y`.

Questions :

1. représenter graphiquement les quatre cas possibles dans l'étude de f(`w`) ;
2. montrer que cet algorithme s'arrête toujours après un nombre fini d'itérations qu'on peut calculer à l'avance, en fonction de a, b et ϵ, si on suppose que x vaut $a + (b - a)\frac{\sqrt{5}-1}{2}$;
3. montrer que les conditions énoncées sont bien vérifiées par le résultat produit par l'algorithme ;
4. programmer une méthode de minimisation par section dorée utilisant les éléments suivants :
   – une classe `Intervalle` pour représenter l'intervalle d'étude ;
   – une interface `Fonction` pour représenter la fonction à minimiser.

Exercice 3.4 :

On propose l'interface suivante pour décrire un polynôme :

```
public interface Polynome {
  public int degré() ;
  public double coeff(int i) ;
}
```

1. Ecrire une méthode de classe calculant la valeur d'un polynôme en un point, par la méthode de Horner et en utilisant l'interface `Polynome`.
2. Proposer une implantation de l'interface `Polynome` se basant sur un tableau de `double`.

3. Proposer une implantation de `Polynome` se basant sur une liste (en utilisant l'interface `List`) de monômes. On commencera par définir une classe `Monome` permettant de stocker un monôme, c'est-à-dire un degré et un coefficient. La liste de monômes ne devra contenir que les monômes de coefficient non nul. La classe `Monome` peut elle implanter l'interface `Polynome` ?

Exercice 3.5 :

On donne l'interface suivante, destinée à représenter des ensembles d'entiers :

```
public interface Ensemble {
  public int cardinal() ;
  public boolean contient(int i) ;
}
```

1. Donner une programmation de cette interface basée sur des listes d'entiers.
2. Donner une programmation de cette interface basée sur un tableau de `boolean`, la case numéro `i` contenant `true` si et seulement si l'entier `i` appartient à l'ensemble.
3. Comparer les deux programmations, en particulier leur avantages et incovénients respectifs.
4. Proposer une interface `EnsembleModifiable` étendant l'interface `Ensemble` afin de permettre l'ajout et la suppression d'éléments de l'ensemble appelant.
5. Modifier les deux classes afin qu'elles implantent `EnsembleModifiable`.
6. Quel problème poserait l'ajout à l'interface `Ensemble` de la méthode ?

   `boolean sousEnsemble(Ensemble e)`

   Cette méthode devrait renvoyer `true` si et seulement si l'ensemble appelant est un sous-ensemble de l'ensemble paramètre.

3.3 Algorithme sous forme d'objet

Exercice 3.6 :

On souhaite représenter sous forme d'objets des algorithmes de transformation de chaînes de caractères. Pour ce faire, on propose l'interface suivante :

```
public interface StringMap {
  public char map(char c) ;
}
```

1. Ecrire une méthode de classe qui à une chaîne de caractères `s` et une référence de type `StringMap f` associe la chaîne de caractères obtenue en appliquant à chaque caractère de `s` la méthode `map` de `f`.
2. Proposer une classe `ToUpper` implantant `StringMap` et permettant de passer en majuscule une chaîne de caractères.
3. Proposer une classe `Replace` implantant `StringMap` et permettant de remplacer un caractère par un autre.
4. Proposer une classe `Cesar` implantant `StringMap` et permettant de réaliser un codage césarien. On rappelle que ce codage s'obtient en décalant les caractères de `x` positions dans l'alphabet (`x` est la clé du code). Ajouter à cette classe une méthode `decode` qui fabrique un nouvel objet `Cesar` réalisant le décodage correspondant au codage proposé par l'objet appelant.

Exercice 3.7 :

On reprend l'exercice précédent en definissant une interface `StringMapAndCut` qui étend l'interface `StringMap` de la façon suivante :

```
public interface StringMapAndCut extends StringMap {
  public boolean keep(char c) ;
}
```

Le principe de la méthode `keep` est simple. Avant de transformer le caractère de la chaîne de départ grâce à la méthode `map`, on vérifie par `keep` qu'on doit bien garder le caractère. Si c'est le cas, on le transforme par `map`. Sinon, on passe directement au caractère suivant.

1. Programmer une méthode de classe qui réalise la transformation qui vient d'être décrite.
2. Proposer une classe `ToNumber` implantant `StringMap` et permettant de ne conserver que les chiffres d'une chaîne de caractères.
3. Proposer une classe `ToUpperLetter` implantant `StringMap` et permettant de ne conserver que les lettres d'une chaîne de caractères en les passant en majuscules.