prosper class: page intentionnally left blank to overcome an incompatibility bug between B. Gaulle ’french’ package and the seminar class.

Compléments de Java

Serge Rosmorduc

rosmord@iut.univ-paris8.fr

IUT de Montreuil

Université de Paris 8

Interfaces

- En java, l'héritage signifie « est-un » ;
- Pas d'héritage multiple $\Rightarrow$ l'héritage est une ressource précieuse ;
- Une interface permet de dire qu'une classe « sait faire » quelque chose, est non plus « est un » quelque chose ;
- exemple : `ActionListener` $\Rightarrow$ peut réagir à un événement.

Interfaces (1)

- En java, une interface est un ensemble de méthodes et de constantes.

Interfaces (1)

- En java, une interface est un ensemble de méthodes et de constantes.
- l'écriture d'une interface ressemble à celle d'une classe, mais sans corps de méthodes :

Interfaces (1)

- En java, une interface est un ensemble de méthodes et de constantes.
- l'écriture d'une interface ressemble à celle d'une classe, mais sans corps de méthodes :

```
public interface FonctionDUneVariable {
  //retourne vrai si la fonction est définie en x
  boolean estDefinie(double x);
  //calcule la valeur de la fonction en x :
  double calculer(double x);
}
```

Interfaces (2)

- Une classe *implémente* une interface X ssi :
  - elle déclare qu'elle le fait au moyen du mot-clef `implements`

  et
  - toutes les méthodes de X sont définies pour la classe

  ou
  - elle hérite d'une classe qui implémente X
- Une classe peut implémenter **plusieurs interfaces**

Interfaces (3)

```
public class Sinus
   implements FonctionDUneVariable{
        public boolean estDefinie(double x)
        {
                return true;
        }
        public double calculer(double x)
        {
                return Math.sin(x);
        }
}
```

Interfaces (4): utilisation

on peut déclarer un handle vers une interface I et lui assigner un objet d'une classe qui implémente I :

```
public class Courbe {
  private FonctionDUneVariable f;
  public Courbe(FonctionDUneVariable f) {
    this.f= f}
  public void tracer() {
    for(int i= 0; i<100; i++)
     System.out.println(""+ i+ " => "+ f.calculer(i));
  }
  public static void main(String a[]) {
    Sinus s= new Sinus();
    Courbe c= new Courbe(s); c.tracer();
  }
}
```

Interfaces (5): constantes

Tout champ d'une interface est considéré comme une constante (`public static final` est sous-entendu) :

```
public interface CartesConstantes {
    int AS= 1;
    int VALET= 11;
    int DAME= 12;
    int ROI= 13;
    int JOCKER= 14;
    int CARREAU= 1;
    int TREFLE = 2;
    int COEUR= 3;
    int PIQUE= 4;
}
```

Interfaces (6): constantes

```
public class JeuDeCartes implements CartesConstantes {
    int valeur; /* prend ses valeurs dans 1..14 */
    int couleur;
    ...
    public String toString() {
        String resultat;
        if (valeur == AS) {resultat= "as";}
        else if (valeur == VALET) {resultat= "valet";}
        ...
        if (couleur == CARREAU)
            resultat+= " de carreau";
        else if ...
        ...
        return resultat;
    }
}
```

Conversions de types (*cast*)

- certaines affectations sont *impossibles* car illogiques :

```
Etudiant e= new JPanel();
```

Conversions de types (*cast*)

- certaines affectations sont *impossibles* car illogiques :

```
Etudiant e= new JPanel();
```

Conversions de types (*cast*)

- certaines affectations sont *impossibles* car illogiques :

```
Etudiant e= new JPanel();
```

- D'autres fonctionnent naturellement :
  - pour les types de base, les affectations sans perte de précision (ex. : affectation d'un `int` dans un `double`) ;
  - pour les classes, `A = B`, où `A` est une classe parente de `B` (car *tout B est un A*) ;

Conversions de types (*cast*)

- certaines affectations sont *impossibles* car illogiques :

```
Etudiant e= new JPanel();
```

- D'autres fonctionnent naturellement :
  - pour les types de base, les affectations sans perte de précision (ex. : affectation d'un `int` dans un `double`) ;
  - pour les classes, `A = B`, où `A` est une classe parente de `B` (car *tout B est un A*) ;
- Pour le reste : conversion explicite *(cast)*.

Cast: types de base

- Certaines conversions sont naturelles :

```
int i= 1;
double x= i;
```

- D'autres se font avec perte d'information :

```
double y= 3.15;
int b= y; ????
```

Cast: types de base

- Certaines conversions sont naturelles :

```
int i= 1;
double x= i;
```

- D'autres se font avec perte d'information :

```
double y= 3.15;
int b= (int) y;
```

Cast : objets

- Soient quatre classes `A`, `B`, `C`, `D` avec `B extends A` et `C extends A`. `D` n'a rien à voir avec `A`. Alors :

```
A a; D d = new D();
B b= new B(); C c= new C();
```

Cast : objets

- Soient quatre classes `A`, `B`, `C`, `D` avec `B extends A` et `C extends A`. `D` n'a rien à voir avec `A`. Alors :

```
A a; D d = new D();
B b= new B(); C c= new C();
a= b; // légal car A sur-classe de B
```

Cast : objets

- Soient quatre classes `A`, `B`, `C`, `D` avec `B extends A` et `C extends A`. `D` n'a rien à voir avec `A`. Alors :

```
A a; D d = new D();
B b= new B(); C c= new C();
a= b; // légal car A sur-classe de B
b= a; // illégal.
```

Cast : objets

- Soient quatre classes `A`, `B`, `C`, `D` avec `B extends A` et `C extends A`. `D` n'a rien à voir avec `A`. Alors :

```
A a; D d = new D();
B b= new B(); C c= new C();
a= b; // légal car A sur-classe de B
b= a; // illégal.
b= (B)a; // légal.
```

Cast : objets

- Soient quatre classes `A`, `B`, `C`, `D` avec `B extends A` et `C extends A`. `D` n'a rien à voir avec `A`. Alors :

```
A a; D d = new D();
B b= new B(); C c= new C();
a= b; // légal car A sur-classe de B
b= a; // illégal.
b= (B)a; // légal.
a= c; // légal
```

Cast : objets

- Soient quatre classes `A`, `B`, `C`, `D` avec `B extends A` et `C extends A`. `D` n'a rien à voir avec `A`. Alors :

```
A a; D d = new D();
B b= new B(); C c= new C();
a= b; // légal car A sur-classe de B
b= a; // illégal.
b= (B)a; // légal.
a= c; // légal
b= (B)a; // légal. échec à l'exécution
```

Cast : objets

- Soient quatre classes `A`, `B`, `C`, `D` avec `B extends A` et `C extends A`. `D` n'a rien à voir avec `A`. Alors :

```
A a; D d = new D();
B b= new B(); C c= new C();
a= b; // légal car A sur-classe de B
b= a; // illégal.
b= (B)a; // légal.
a= c; // légal
b= (B)a; // légal. échec à l'exécution
a= d; // complètement illégal
```

Cast : objets

- Soient quatre classes `A`, `B`, `C`, `D` avec `B extends A` et `C extends A`. `D` n'a rien à voir avec `A`. Alors :

```
A a; D d = new D();
B b= new B(); C c= new C();
a= b; // légal car A sur-classe de B
b= a; // illégal.
b= (B)a; // légal.
a= c; // légal
b= (B)a; // légal. échec à l'exécution
a= d; // complètement illégal
```

- **On peut affecter un élément x de classe X à un handle d'une classe Y ssi X étend Y (pas de cast) ou Y étend X (cast). Dans le dernier cas, il *faut* que x soit réellement de classe Y.**

Cast d'objet : exemple

On veut implémenter l'interface suivante :

Comparable

```
// Une classe qui implémente cette interface est
// dotée d'une relation d'ordre.
public interface Comparable {
  // Renvoie un nombre
  // -- négatif si this < o
  // -- nul si this = o
  // -- positif si this > o
  int CompareTo(Object o);
}
```

Cast : objets

```
public class Rationnel implements Comparable {
  private int p, q; // On suppose q > 0
  ...
  // a priori, l'argument de compareTo
  // sera vraiment un rationnel, mais il est vu
  // par la fonction comme un objet.
  public int compareTo(Object o) {
    // Ne fonctionne que
    //   si o est de type Rationnel
    Rationnel r= (Rationnel)o;
    // signe de a/b - c/d = signe de
    //   ad -bc car b et d > 0
    return (p*r.q - q*r.p);
```

La classe Vector

- Problème : créer des listes/ des tableaux extensibles.
- Solution simple et rapide : utiliser la classe Vector.
- Un Vector est un tableau d'**Objects**
- méthodes choisies :

  void add(Object o) : ajoute o à la fin du Vector ;

  int size() : retourne le nombre d'éléments du vecteur ;

  void set(int i, Object o) : remplace le i^e élément par o ;

  Object get(int i) : récupère le i^e élément ;

  boolean remove(Object o) : enlève o du vecteur ;

  boolean remove(int i) : enlève le le i^e élément ;

Vector : utilisation

```
import java.util.Vector;

public class TestVector {
    public static void main(String a[]) {
        Vector v= new Vector();
        v.add(new Etudiant("Turing"));
        v.add(new Etudiant("Babbage"));
        // ...
        for (int i= 0; i < v.size(); i++)
            {
                String nom;
                nom= ((Etudiant)v.get(i)).getNom();
                System.out.println(nom);
            }
    }
}
```

Identité des objets (1)

- Par défaut, deux objets sont égaux s'ils ont la même adresse :

```
String a= "hello"; String b= "hello";
if (a == b)
  System.out.println("a == b");
else
  System.out.println("a != b");
```

- l'opérateur « `==` » teste toujours l'égalité des adresses ;
- la méthode `boolean equals(Object)`, définie dans la classe `Object` :
  - teste par défaut l'égalité des adresses ;
  - peut être redéfinie. ex. : dans la classe `String`.

Identité des objets (2)

```
public class Point {
  private double x, y;
  ...
  public boolean equals(Object o) {
    if (o instanceOf Point) {
      return ((Point)o).x == x &&
             ((Point)o).y == y;
    else
      return false;
    }
  }
  ...
}
```

Identité des objets (3)

- très souvent, l'égalité par adresse est suffisante (ex. : JPanel) ;
- en toute rigueur, quand on redéfinit la méthode `equals`, il est nécessaire de redéfinir la méthode `hashcode`.