Héritage, classes abstraites et interfaces

Université Française d'Egypte
Version O 9.6 – 12/11/20
Richard Grin

1

Plan de cette partie

- Héritage
- Classe abstraite
- Interface
- Réutiliser des classes

2

Héritage

3

Réutilisation

- Intéressant de pouvoir réutiliser du code (déjà écrit, déjà testé)

4

Réutilisation par classe cliente

- Une classe `C` veut utiliser/réutiliser la classe `A`
- `C` crée une instance de `A` et lui demande un service
- `C` est une classe cliente de la classe `A` ; on dit aussi que la classe `C` dépend de la classe `A`
- Exemple : la classe `Livre` du TP1 est cliente de la classe `Comptable` ; pour fonctionner, elle dépend de la classe `Comptable`

5

Réutilisation avec modifications

- On souhaite modifier en partie le comportement de `A` avant de la réutiliser :
  - modifier une méthode/un comportement
  - ajouter une méthode/une fonctionnalité

6

Réutilisation avec modifications du code source

- Copier le code source de `A` dans une classe `B` et ensuite modifier le code de `B`
- Problèmes :
  - modifier le code écrit pour `A`, peut introduire des bugs dans ce qui fonctionnait avant
  - améliorations futures du code de `A` ne seront pas dans la classe `B`. Difficile à maintenir !
  - code source de `A` pas toujours disponible

7

Réutilisation par l'héritage

- L'héritage permet d'écrire une classe `B`
  - qui se comporte dans les grandes lignes comme la classe `A`
  - mais avec quelques différences

  sans toucher ni copier le code source de `A`

8

Vocabulaire

- La classe `B` qui hérite de la classe `A` s'appelle une classe fille ou sous-classe
- La classe `A` s'appelle une classe mère, classe parente ou super-classe

9

Exemple d'héritage - classe mère

```
public class Rectangle {
  private int x, y; // sommet en haut à gauche
  private int largeur, hauteur;
  // Constructeurs,
  // méthodes getX(), setX(int)
  // getHauteur(), getLargeur(),
  // setHauteur(int), setLargeur(int),
  // contient(Point), intersecte(Rectangle)
  // translateToi(Vecteur), toString(),...
  . . .
  public void dessineToi(Graphics g) {
    g.drawRect(x, y, largeur, hauteur);
  }
}
```

Méthodes `public`

g représente une zone de l'écran sur laquelle on peut dessiner

10

Exemple d'héritage - classe fille

```
public class RectangleColore extends Rectangle {
  private Color couleur; // nouvelle variable
  // Constructeurs
  . . .
  // Nouvelles Méthodes
  public Color getCouleur() { return this.couleur; }
  public void setCouleur(Color c) {this.couleur = c; }
  // Méthode modifiée
  public void dessineToi(Graphics g) {
    g.setColor(couleur);
    g.fillRect(getX(), getY(),
               getLargeur(), getHauteur());
  }
}
```

Utilisation des accesseurs. Pourquoi ?

11

Exemples d'héritage

- Classe mère `Vehicule` ;
  classes filles `Velo`, `Voiture` et `Camion`
- Classe fille `Avion` ;
  classes mères `ObjetVolant` et `ObjetMotorise`
- Classe fille `Polygone` ;
  classe mère `FigureGeometrique`

12

2 façons de voir l'héritage

- Particularisation-généralisation : une classe fille est un type particulier de la classe mère
  - un polygone *est une* figure géométrique, mais une figure géométrique particulière
  - la notion de figure géométrique est une généralisation de la notion de polygone
- Une classe fille offre des services nouveaux ou enrichis : la classe `RectangleColore` permet de dessiner avec des couleurs et pas seulement en « noir et blanc »

L'héritage en Java

- En Java, une classe ne peut avoir qu'une seule classe mère (pas d'héritage multiple)
- `extends` indique la classe mère :
  `class RectangleColore extends Rectangle`
- Par défaut, une classe hérite de la classe `java.lang.Object`
- Ce qui implique que la classe `Object` est la classe ancêtre de toutes les classes

Exemples d'héritage

- `class Voiture extends Vehicule`
- `class Velo extends Vehicule`
- `class Polygone extends FigureGeometrique`

Ce que peut faire une classe fille

- La classe qui hérite peut
  - ajouter des variables d'état, des méthodes, des constructeurs
  - redéfinir des méthodes (même signature)
  - surcharger des méthodes (même nom, mais pas même signature)
- Elle ne peut retirer aucune variable d'état ou méthode

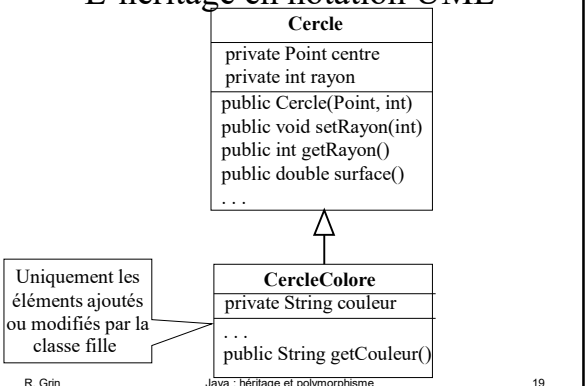
Principe important lié à la notion d'héritage

- Si « `B extends A` », le grand principe est que tout `B` est un `A`
- Par exemple, un rectangle coloré *est un* rectangle ; une voiture *est un* véhicule
- Ne pas utiliser l'héritage pour réutiliser du code si le principe « est-un » n'est pas vérifié

Héritage et typage

- Définition : `B` est un sous-type de `A` si on peut ranger une expression de type `B` dans une variable de type `A`
- `A a = new B(…);` est autorisé
- Si `B` hérite de `A` , `B` est un sous-type de `A`
- En effet, selon le principe « tout `B` est un `A` », on peut ranger un `B` dans une variable de type `A`
- `Vehicule v = new Velo();`

L'héritage en notation UML



19

Compléments sur les constructeurs d'une classe

20

1ère instruction d'un constructeur

- La première instruction d'un constructeur peut être un appel
  - à un autre constructeur de la classe : `this(...)`
  - à un constructeur de la classe mère : `super(...)`
- Interdit de mettre `this(...)` et `super(...)` ailleurs qu'au début d'un constructeur

21

Constructeur de la classe mère

```
public class Rectangle {
  private int x, y, largeur, hauteur;

  public Rectangle(int x, int y,
                   int largeur, int hauteur) {
    this.x = x;
    this.y = y;
    this.largeur = largeur;
    this.longueur = longueur;
  }
  . . .
}
```


22

Constructeurs de la classe fille

```
public class RectangleColore extends Rectangle {
  private Color couleur;

  public RectangleColore(int x, int y,
                         int largeur, int hauteur
                         Color couleur) {
    super(x, y, largeur, hauteur);
    this.couleur = couleur;
  }

  public RectangleColore(int x, int y,
                         int largeur, int hauteur) {
    this(x, y, largeur, hauteur, Color.black);
  }
```

Couleur par défaut

23

Appel implicite du constructeur de la classe mère

- Si la première instruction d'un constructeur n'est ni `super(...)`, ni `this(...)`, le compilateur ajoute au début du constructeur un appel `super()` au constructeur sans paramètre de la classe mère

Erreur de compilation s'il n'existe pas !

⇒ Un constructeur de la classe mère est toujours exécuté avant les autres instructions du constructeur

24

Exemple

```
public class A {
  public A() { System.out.print("A"); }
}
public class B extends A {
  public B() { System.out.print("B"); }
}
public class C extends B {
  public C() { System.out.print("C"); }
}
public class Main {
  public static void main(String[] args) {
    C c = new C();
  }
}
```

`java Main` affiche quoi ?

25

Exemple

```
public class A {
  public A() { super(); System.out.print("A"); }
}
public class B extends A {
  public B() { super(); System.out.print("B"); }
}
public class C extends B {
  public C() { super(); System.out.print("C"); }
}
public class Main {
  public static void main(String[] args) {
    C c = new C();
  }
}
```


26

Toute première instruction exécutée par un constructeur

- La première instruction d'un constructeur de la classe mère est l'appel à un constructeur de la classe « grand-mère », et ainsi de suite...
- Quelle est la toute première instruction qui est exécutée par un constructeur ?
- Constructeur (sans paramètre) de la classe `Object` !

27

Complément sur le constructeur par défaut d'une classe

- Ce constructeur par défaut n'appelle pas explicitement un constructeur de la classe mère

  ⇒ un appel du constructeur sans paramètre de la classe mère est automatiquement effectué

28

Question…

```
class A {
  private int i;

  A(int i) {
    this.i = i;
  }

}
```

Compile ?
S'exécute ?

```
class B extends A { }
```


29

Exemples de constructeurs (1)

```
public class Cercle {
  // Constante
  public static final double PI = 3.14;
  // Variables
  private Point centre;
  private int rayon;
  // Constructeur
  public Cercle(Point c, int r) {
    centre = c;
    rayon = r;
  }
  ...
```

⇒ le compilateur n'ajoute pas le constructeur sans paramètre par défaut

Appel implicite du constructeur `Object()`

30

Exemples de constructeurs (2)

```
public class CercleColore extends Cercle {
  private String couleur;
  public CercleColore(Point p, int r, String c) {
    super(p, r);
    couleur = c;
  }
  public void setCouleur(String c) {
    couleur = c;
  }
  public String getCouleur() {
    return couleur;
  }
}
```

Que se passe-t-il si on enlève cette instruction ?

31

Grave erreur de débutant !

```
public class CercleColore extends Cercle {
  private Point centre;
  private int rayon;
  private String couleur;
  public CercleColore(Point p, int r, String c) {
    centre = p;
    rayon = r;
    couleur = c;
  }
```

Quelle est l'erreur ?

à remplacer par quoi ?

`centre` et `rayon` sont hérités de `Cercle` ; ne pas les redéclarer !

32

Accès aux membres hérités Protection `protected`

33

Accès à la classe mère - `protected`

- Une classe fille n'a pas accès aux membres `private` de sa classe mère
- La protection `protected` ouvre l'accès aux classes filles

34

Exemple d'utilisation de `protected`

```
public class Animal {
  protected String nom;
  . . .
}
```

```
public class Poisson extends Animal {
  private int profondeurMax;

  public Poisson(String unNom, int uneProfondeur) {
    nom = unNom;        // utilisation de nom
    profondeurMax = uneProfondeur;
  }
```


35

`protected` et paquetage

- `protected` donne aussi accès aux classes du même paquetage

36

Toutes les protections d'accès

- Dans l'ordre croissant de protection :
  - `public`
  - `protected`
  - *package* (protection par défaut)
  - `private`
- `protected` est moins restrictive que la protection par défaut !

37

Protections des variables

- Éviter les variables d'instance `protected` car elles nuisent à l'encapsulation de la classe par rapport à ses classes filles
- La maintenance de la classe mère est plus complexe : on ne peut pas changer sa structure de données puisque des classes filles peuvent avoir déjà utilisé ces variables
- Pas ce problème avec les méthodes `protected`

38

Classe `Object`

39

Classe `Object`

- La racine de l'arbre d'héritage des classes est la classe `java.lang.Object`
- Pas de variables d'état (ni d'instance, ni de classe), pas de méthodes `static`
- Les méthodes d'instance de `Object` sont héritées par toutes les classes
- Les plus utilisées sont les méthodes `toString`, `equals`, `hashcode` et `getClass`

40

Méthode `equals`

- `public boolean equals(Object obj)`
  retourne `true`
  ⇔
  `this` a « la même valeur » que `obj`

Dans quelle classe avez-vous déjà vu cette méthode ?

- L'implémentation de `Object` retourne `true`
  ⇔
  `this` référence le même objet que `obj`
- Peut être redéfinie dans les classes qui ont une relation d'égalité (d'équivalence)

41

Exemple de `equals` et `toString`

```
public class Fraction {
  private int num, den;
  . . .
  public String toString() {
    return num + "/" + den;
  }
  public boolean equals(Object o) {
    if (! (o instanceof Fraction))
      return false;
    return num * ((Fraction)o).den
        == den * ((Fraction)o).num;
  }
```

Que se passe-t-il si on enlève ce cast ?

`a/b = c/d` *ssi* `a*d = b*c`

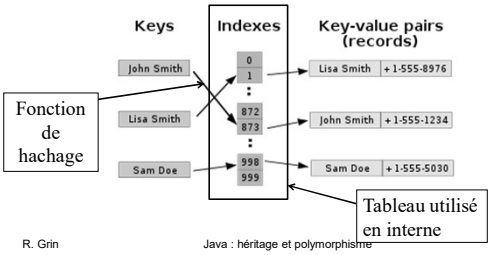
42

Problème fréquent

- Rechercher un objet en connaissant une clé qui l'identifie
- Par exemple, chercher les informations sur l'employé qui a le matricule AF823
- Une table de hachage (*hash table*) permet d'implémenter cette fonctionnalité, avec de très bonnes performances, même si la recherche se fait parmi des millions d'objets

Table de hachage

- Structure de données qui permet de retrouver très rapidement un objet si on connaît sa clé



Exemple de fonction de hachage

- À un nom on fait correspondre la somme des positions dans l'alphabet des lettres du nom
- Par exemple, toto → 20 + 15 + 20 + 15 = 70
- Le nom otto donne la même valeur
- Cette fonction de hachage n'est pas bonne car elle donne trop de valeurs égales pour des noms différents ; on pourrait l'améliorer en faisant intervenir dans le calcul la position de chaque lettre

Méthode `hashCode`

- `public int hashCode()`
  utilisée comme fonction de hachage dans les tables de hachage utilisées par le JDK
- La méthode `hashCode` de 2 objets égaux au sens de `equals` doit retourner le même entier
- Toute classe qui redéfinit `equals` doit donc obligatoirement redéfinir `hashCode`

Exemple de `hashCode` dans `Fraction`

```
  /** Réduit la fraction et applique une
     formule magique ;-) */
  public int hashCode() {
    Fraction f = reduire();
    return 31 * (31 * 17 + f.den) + f.num;
  }
}
```


Méthode `getClass` - classe `java.lang.Class`

- `public Class getClass()`
  retourne la classe de l'objet
- Une instance de `Class` représente une classe utilisée par l'application
- La méthode `getName()` de la classe `Class` retourne le nom complet de la classe

instanceof

- Si b est une instance d'une sous-classe `B` de `A`, `b instanceof A` est égal à `true`
- Tester si un objet o est de la même classe que l'objet courant :

```
if (o != null &&
    o.getClass() == this.getClass())
```


Compléments sur la redéfinition d'une méthode

Annotation pour la redéfinition

- Conseillé d'annoter les méthodes qui redéfinissent une méthode

```
@Override
public Dimension getPreferredSize() {
```

- Pour la lisibilité et pour repérer des fautes de frappe dans le nom de la méthode

Redéfinition et surcharge

- Une méthode redéfinit une méthode héritée quand elle a
  la même signature que l'autre méthode
- Une méthode surcharge une méthode quand elle a
  le même nom, mais pas la même signature, que l'autre méthode

Exemple de redéfinition

- Redéfinition de la méthode de la classe `Object` « `boolean equals(Object)` »

```
public class Entier {
  private int i;
  public Entier(int i) {
    this.i = i;
  }
  @Override
   public boolean equals(Object o) {
    if (o == null || (o.getClass() != this.getClass()))
       return false;
    return i == ((Entier)o).i;
  }
}
```

Cette classe vous rappelle une classe du JDK ?

Peut-on enlever cette instruction `if` ?

Exemple de surcharge

- Surcharge de la méthode `equals` de `Object` :

```
public class Entier {
  private int i;
  public Entier(int i) {
    this.i = i;
  }
  public boolean equals(Entier e) {
    if (e == null) return false;
    return i == e.i;
  }
}
```

Il faut redéfinir la méthode `equals` et ne pas la surcharger

super.

- Soit `B` une classe fille de `A`
- Dans une méthode d'instance `m` de `B`, « `super.` » sert à désigner un membre de `A`, en particulier une méthode redéfinie dans `B`
- Quelle valeur va retourner cette méthode de `B` ?

```
@Override
public int m(int i) {
  return 500 + super.m(i);
}
```


55

Polymorphisme

56

Une question…

- `B` hérite de `A`
  `B` redéfinit une méthode `m()` de `A`
- `A a = new B(5);`
  `a.m();`

`a` référence un `B` mais est déclarée du type `A`

Compile ?

- Quelle méthode `m` est exécutée, celle de `A` ou celle de `B` ?
- La méthode appelée ne dépend que du type réel de l'objet qui reçoit le message (pas du type déclaré). C'est donc…

la méthode `m` de `B` qui est exécutée

57

Polymorphisme

- Polymorphisme car une même écriture peut correspondre à plusieurs appels de méthodes selon les exécutions
- Signature de `f` : `A f()`
- `A a = x.f();`
  `a.m();`

`f` peut renvoyer une instance de `A` ou de n'importe quelle sous-classe de `A`

appelle la méthode `m` de quelle classe ?

- Réponse : méthode `m` de `A`
  ou d'une sous-classe de `A` (on ne connaitra la méthode qu'à l'exécution)

58

Mécanisme du polymorphisme

- Polymorphisme obtenu grâce au « *late binding* » (liaison retardée) : la méthode qui est exécutée est déterminée à l'exécution (pas dès la compilation)

59

Polymorphisme

- Notion fondamentale de la programmation objet, indispensable pour une utilisation efficace de l'héritage

60

Exemple de polymorphisme

```
public class Figure {
  public void dessineToi() { }
}
```
Méthode vide

```
public class Rectangle extends Figure {
  @Override
  public void dessineToi() {
    . . .
}
```

```
public class Cercle extends Figure {
  @Override
  public void dessineToi() {
    . . .
}
```


61

Exemple de polymorphisme (suite)

```
public class Dessin { // dessin composé de plusieurs figures
  private Figure[] figures;
  . . .
  public void afficheToi() {
    for (int i = 0; i < nbFigures; i++)
      figures[i].dessineToi();
  }
  public static void main(String[] args) {
    Dessin dessin = new Dessin(30);
    . . . // création des points centre, p1, p2
    dessin.ajoute(new Cercle(centre, rayon));
    dessin.ajoute(new Rectangle(p1, p2));
    dessin.afficheToi();
    . . .
```

dessineToi() de quelle classe sera exécutée ?

dessineToi() du type réel de figures[i]

ajoute une figure dans figures

62

Typage statique et polymorphisme

- La classe `Figure` doit posséder une méthode `dessineToi()`, sinon, le compilateur refuse de compiler `figures[i].dessineToi()`
- Ce typage statique garantit dès la compilation l'existence de la méthode appelée

Pourquoi ?

63

Utilisation du polymorphisme

- Évite les codes qui comportent de nombreux embranchements et tests
- Sans polymorphisme, la méthode `dessineToi` aurait dû s'écrire :

```
for (int i = 0; i < nbFigures; i++) {
  if (figures[i] instanceof Rectangle) {
    . . . // dessiner un rectangle
  }
  else if (figures[i] instanceof Cercle) {
    . . . // dessiner un cercle
  }
}
```


64

Utilisation du polymorphisme (2)

- Facilite l'extension des programmes : il suffit de créer de nouvelles sous-classes sans toucher au code source déjà écrit
- Si on ajoute une classe `Losange`, le code de `afficheToi` sera toujours valable
- Sans polymorphisme, il aurait fallu modifier le code source de la classe `Dessin` pour ajouter un nouveau test :

```
if (figures[i] instanceof Losange) {
  . . . // dessiner un losange
```


65

Extensibilité

- En programmation objet, une application est dite extensible si on peut étendre ses fonctionnalités
  sans toucher au code source déjà écrit

66

Mécanisme de la liaison retardée

- `o` instance de la classe `C`
- Quelle classe contient la définition de la méthode exécutée par le code « `o.m()` » ?
- Classe `C` si elle contient la définition d'une méthode `m()`

Possible de remonter jusqu'à `Object` sans trouver `m` ?

- Sinon, la recherche de `m()` se poursuit dans la classe mère de `C`, puis dans la classe mère de cette classe mère, et ainsi de suite, jusqu'à trouver la définition d'une méthode `m()`

Cast (transtypage)

Vocabulaire

- Classe réelle (ou type réel) d'un objet : classe du constructeur qui a créé l'objet
- Type déclaré d'un objet : type de la variable qui référence l'objet

Cast : conversions de classes

- *cast* = forcer le compilateur à considérer un objet comme étant d'un certain type
- Seuls *casts* autorisés :
  casts vers classe mère ou classe fille

Syntaxe

- Caster un objet `o` en classe `C` :
  `(C) o`
- Exemple :

```
Velo v = new Velo();
Vehicule v2 = (Vehicule) v;
```


Vocabulaire

- *upcast* et de *downcast* car la classe mère est souvent dessinée au-dessus de ses classes filles

UpCast : classe fille → classe mère

- Caster vers une classe ancêtre de la classe réelle, par exemple `Velo` vers `Vehicule`
- Toujours possible, à cause de la relation *est-un* de l'héritage
- Souvent implicite

73

Utilisation du *UpCast*

- Pour profiter ensuite du polymorphisme :

```
Figure[] figures = new Figure[10];
figures[0] = (Figure) new Cercle(p1, 15);
. . .
figures[i].dessineToi();
```


74

Utilisation du *UpCast*

- Pour profiter ensuite du polymorphisme :

```
Figure[] figures = new Figure[10];
figures[0] = new Cercle(p1, 15);
. . .
figures[i].dessineToi();
```


75

DownCast : classe mère → classe fille

- Caster vers une classe fille de sa classe de déclaration, par exemple `Vehicule` vers `Velo`
- Toujours accepté par le compilateur
- Mais peut provoquer une erreur à l'exécution si l'objet n'est pas du type de la classe fille
- Toujours explicite

76

Utilisation du *DownCast*

- Utilisé pour appeler une méthode de la classe fille qui n'existe pas dans une classe ancêtre

```
Figure f1 = new Cercle(p, 10);
. . .
Point p1 = ((Cercle) f1).getCentre();
```

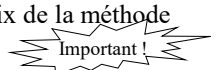
Parenthèses obligatoires car « . » a une plus grande priorité que le cast

Si on ne met pas ces parenthèses, ça compile ? ça s'exécute ?

77

cast et *late binding*

- Un *cast* ne modifie pas le choix de la méthode exécutée
- Celle-ci est déterminée par le type réel de l'objet qui reçoit le message



78

Contraintes pour les déclarations des méthodes redéfinies

79

Raison des contraintes

- Si `B` hérite de `A`, toute instance de `B` doit pouvoir être considérée comme une instance de `A`
- Donc, si on a la déclaration
  `A a;`

  Toute expression où intervient la variable `a` doit pouvoir être compilée et exécutée si `a` contient une instance de `B`
- Les 2 contraintes suivantes de cette section découlent de cette contrainte

80

Covariance du type retour en Java

- Le plus souvent le type retour d'une méthode redéfinie est le même que le type retour `T` de la méthode redéfinie
- Cependant, il est possible, dans la sous-classe, de remplacer le type retour de la méthode redéfinie par une sous-classe de `T` (covariance)

81

Exemple covariance du type retour

- La méthode de la classe `Object`
  `Object clone()`
  peut être redéfinie en
  `C clone()`
  dans une sous-classe `C` de `Object`

82

Contrainte sur l'accessibilité

- La redéfinition ne doit jamais être moins accessible
- Par exemple, la redéfinition d'une méthode `public` ne peut pas être `protected`

Est-ce qu'une méthode `private` peut être redéfinie en une méthode `protected` *?*

83

Compléments sur l'héritage

84

Méthode `static`

- On ne peut pas redéfinir une méthode `static`
- Pas de liaison retardée ni de polymorphisme avec les méthodes `static` : la méthode exécutée est déterminée à la compilation, par le type déclaré

85

`final`

- Classe `final` : ne peut pas avoir de classes filles (`String` est `final`)
- Méthode `final` : ne peut pas être redéfinie
- Variable (locale ou d'état) `final` : la valeur ne peut pas être modifiée après son initialisation
- Paramètre `final` : la valeur ne peut pas être modifiée dans le code de la méthode

86

Tableaux et sous-typage

- Les tableaux héritent de la classe `Object`
- Si `B` est un sous-type de `A`, `B[]` est un sous-type de `A[]`
- Exemple : `Cercle[]` est un sous-type de `Figure[]`
- On peut donc écrire :
  `Figure[] tbFig = new Cercle[5];`

87

Problème de typage avec les tableaux

```
Figure fig = new Carre(p1, p2);
Figure[] tbFig = new Cercle[5];
tbFig[0] = fig;
```

Compile ?

Et à l'exécution ?

- Erreur `ArrayStoreException`

Quelle ligne provoque l'erreur et pourquoi ?

88

Classe abstraite

89

Méthode abstraite

- Méthode sans implémentation :

```
public abstract int m(String s);
```

- `m` sera implémentée par les classes filles

90

Classe abstraite

- Une classe doit être déclarée abstraite (`abstract class`) si elle contient une méthode abstraite
- Interdit de créer une instance d'une classe abstraite

91

Complément

- Pourquoi une méthode `static` ne peut pas être abstraite ?

92

Interface

93

Définition

- « Classe » purement abstraite dont toutes les méthodes sont abstraites et publiques
- Une interface peut aussi contenir des méthodes `static` et des méthodes par défaut (`default`)
- Elle peut aussi contenir des méthodes `private`, pour que 2 méthodes `default` puissent partager du code, sans que ce code soit visible à l'extérieur de l'interface

94

Exemples d'interfaces (1/2)

```
public interface Figure {
  public abstract void dessineToi();
  public abstract void deplaceToi(int x, int y);
  public abstract Position getPosition();
}
```

`public abstract` peut être implicite

95

Exemples d'interfaces (1/2)

```
public interface Figure {
  void dessineToi();
  void deplaceToi(int x, int y);
  Position getPosition();
}
```


96

Exemples d'interfaces (2/2)

```
public interface Comparable {
  /** retourne vrai si this est « plus grand » que o */
  boolean plusGrand(Object o);
}
```


97

Classe qui implémente une interface

- `public class C implements I1 { … }`
- 2 cas possibles :
  - soit la classe `C` définit toutes les méthodes abstraites de `I1`
  - soit la classe `C` doit être `abstract`

98

Exemple

```
public class Ville implements Comparable {
  private String nom;
  private int nbHabitants;
  . . .
  @Override
  public boolean plusGrand(Object objet) {
    if (! objet instanceof Ville) {
      throw new IllegalArgumentException(…);
    }
    return nbHabitants > ((Ville)objet).nbHabitants;
  }
}
```

Exactement la même signature que dans l'interface `Comparable`

Pourquoi ce cast ?

Autorisé ?

99

Implémentation de plusieurs interfaces

- Une classe peut implémenter plusieurs interfaces (et hériter d'une classe…) :

```
public class CercleColore extends Cercle
    implements Figure, Coloriable {
  . . .
}
```


100

Contenu des interfaces

- Une interface peut aussi contenir des définitions de constantes publiques (« `public static final` »)
- Les mots clés `public` et `final` peuvent être implicites

101

Accessibilité des interfaces

- Même accessibilité que les classes :
  - `public` : utilisable de partout
  - sinon : utilisable seulement dans le même paquetage

102

Les interfaces comme types de données

- Une interface peut servir à déclarer une variable, un type de base de tableau, un *cast*,…
- `Comparable v1;`
  la classe d'un objet référencé par `v1` doit implémenter l'interface `Comparable`

103

Interfaces et typage

- Si une classe `C` implémente une interface `I`, `C` est un sous-type de `I` :
  `Comparable v = new Ville(…);`

104

Exemple d'interface comme type de données

```
public static boolean croissant(Comparable[] t) {
  for (int i = 0; i < t.length - 1; i++) {
    if (t[i].plusGrand(t[i + 1]))
      return false;
  }
  return true;
}
```

Que fait cette méthode ?

105

Polymorphisme et interfaces

```
public interface Figure {
  void dessineToi();
}
```

```
public class Rectangle implements Figure {
  @Override
  public void dessineToi() {
    . . .
}
```

```
public class Cercle implements Figure {
  @Override
  public void dessineToi() {
    . . .
}
```


106

Cast et interfaces

- On peut faire des *casts* entre une classe et une interface qu'elle implémente :

```
Comparable c1 = new Ville("Cannes", 200_000);
Comparable c2 = new Ville("Nice", 500_000);
. . .
if (c1.plusGrand(c2))
  System.out.println(((Ville)c2).nbHabitant());
```


107

A quoi servent les interfaces ?

- Garantir aux « clients » d'une classe que la classe a certaines méthodes, donc qu'elle peut assurer certains services
- Faire du polymorphisme avec des objets dont les classes n'appartiennent pas à la même hiérarchie d'héritage
- Favoriser la réutilisation :
  `croissant(Comparable[])`

108

Éviter de dépendre de classes concrètes

- Pour rendre plus facile la maintenance d'une application, évitez de faire dépendre une classe de classes concrètes
- Si elle dépend d'interfaces,
  - moins de risques de devoir la modifier pour tenir compte de modifications des dépendances
  - la classe sera plus réutilisable

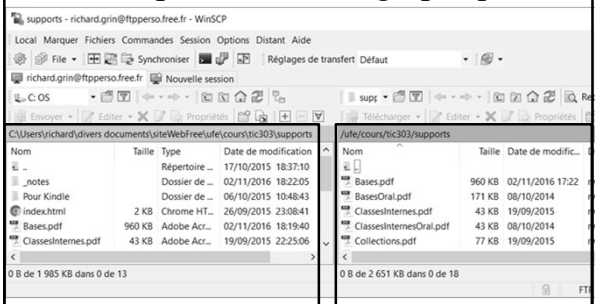
109

Exemple typique

- Une classe « métier » `FTP` est utilisée par une interface graphique `GUI`
- S'il y a un problème pendant le transfert de données, `FTP` passe un message d'erreur à `GUI` pour que l'utilisateur puisse le lire

110

Exemple d'interface graphique



111

Code

```
public class Gui {
  private FTP ftp;

  public Gui() {
    ftp = new Ftp(…);
    ftp.setAfficheur(this);
  }
    . . .
  public void affiche(String m) {...}
```

Interface Utilisateur

Quel est le problème avec ce code?

```
public class Ftp {
  private Gui gui;

  public void setAfficheur(Gui gui) {
    this.gui = gui;
  }
  . . .
  gui.affiche(message);
```

Utilisation de FTP

Comment améliorer ?

`Ftp` ne pourra pas être utilisé avec une autre interface graphique !

112

Code amélioré

```
public class Gui implements Afficheur {
  private Ftp ftp;

  public Gui() {
    ftp = new Ftp(…);
    ftp.setAfficheur(this);
  }
  . . .
  public void affiche(String m) {...}
```

Code de `Afficheur` ?

```
public class Ftp {
  private Afficheur afficheur;

  public void setAfficheur(Afficheur aff) {
    this.afficheur = afficheur;
  }
  . . .
  afficheur.affiche(message);
```


113

Code de l'interface

```
public interface Afficheur {
  void affiche(String message);
}
```


114

Héritage d'interfaces

- Une interface peut hériter (mot-clé `extends`) de plusieurs interfaces :

```
interface I1 extends I2, I3, I4 {
  . . .
}
```

- `I1` hérite de toutes les méthodes et constantes des interfaces `I2`, `I3`, `I4`

Utilité des méthode `default`

- Elles n'existaient pas avant Java 8
- Ce qui rendait difficile de modifier une interface
- L'ajout d'une méthode à l'interface rendait non compilables toutes les classes clientes qui implémentaient l'ancienne version de l'interface

Exemple

- Avant Java 8, pas de méthode `sort` dans l'interface `List<E>`
- Si on ajoutait une telle méthode dans l'interface, toutes les classes non abstraites qui implémentaient l'interface `List<E>` ne compilaient plus car il leur manquait l'implémentation de la méthode `sort`

Quel message d'erreur à la compilation ?

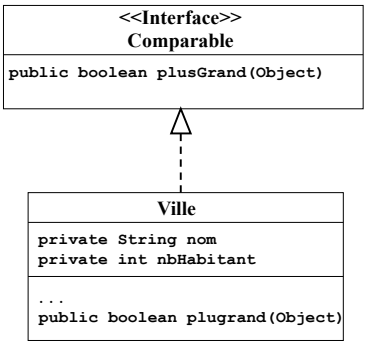
Méthode par défaut

- Une interface peut contenir des méthodes non `abstract`, avec une implémentation
- Il faut ajouter le mot-clé `default` à la méthode : `public default int m() { ... }`
- Elle devient la définition *par défaut* de la méthode pour les classes qui implémentent l'interface

Utilité des méthodes par défaut

- On peut ajouter une nouvelle méthode à une interface déjà implémentée par des classes
- Il suffit de donner une définition par défaut de cette nouvelle méthode, qui sera utilisée par les classes qui implémentent l'interface

Interfaces en notation UML



Faire TP sur héritage

121

Réutilisation

122

Règle pour l'utilisation de l'héritage

- Ne s'en servir que pour représenter la relation « *est-un* » entre classes :
  - un `CercleColore` est un `Cercle`
  - une `Voiture` est un `Vehicule`
- Pourrait aussi être utilisé comme moyen pratique de réutilisation de code : la classe `Parallélépipède` pourrait hériter de la classe `Rectangle` en ajoutant une variable `profondeur`

À éviter !

123

Réutilisation par une classe `C2` du code d'une classe `C1`

- On veut utiliser la classe `C1` pour écrire le code d'une classe `C2`
- Plusieurs moyens :
  - `C2` hérite de `C1`
  - `C2` peut déléguer à une instance de `C1` une partie de la tâche qu'elle doit accomplir

124

Délégation « pure »

- Une méthode `m2()` de la classe `C2` :

```
public int m2() {
  …
  C1 c1 = new C1();
  r = c1.m1();
  …
}
```

création d'une instance de `C1`

utilisation de l'instance

- Variante :

```
public int m2(C1 c1) {
  …
  r = c1.m1();
```


125

Délégation avec composition

```
public class C2 {
  private C1 c1;
  . . .
  public int m2() {
    . . .
    r = c1.m1();
    . . .
  }
```

peut être utilisé à plusieurs occasions ; créé dans le constructeur ou ailleurs

126

Exemples de réutilisation

- Le contour d'une fenêtre dessinée sur l'écran est un rectangle
- Comment réutiliser les méthodes d'une classe `Rectangle` dans la classe `Fenetre` ?

127

Réutilisation par héritage

```
public class Fenetre extends Rectangle {
   ...

  // Hérite de getDimension()

  /** Modifie la taille de la fenêtre */
  public void setDimension(int largeur, int longueur) {
    super.setDimension(largeur, longueur);
    // replacer composants de la fenêtre
    ...
  }
  . . .
}
```


128

Réutilisation par composition

```
public class Fenetre {
   private Rectangle contour;

   /** Retourne la taille de la fenêtre */
   public Dimension getDimension() {
     return contour.getDimension();
   }

   /** Modifie la taille de la fenêtre */
   public void setDimension(int largeur, int hauteur) {
     contour.setDimension(largeur, hauteur);
    // replacer composants de la fenêtre
     ...
   }
   . . .
}
```


129

Inconvénients de l'héritage

- Statique : une classe ne peut hériter de classes différentes à des moments différents
- Souvent difficile de changer une classe mère sans provoquer des problèmes de compatibilité avec les classes filles (mauvaise encapsulation, en particulier si elle a des variables `protected`)
- Pas possible d'hériter d'une classe `final` (comme la classe `String`)

130

Avantages de l'héritage

- Facile à utiliser, car c'est un mécanisme de base du langage
- Souple, car on peut redéfinir facilement les comportements hérités, pour les réutiliser ensuite
- Permet le polymorphisme (mais possible aussi avec les interfaces)
- Facile à comprendre si c'est la traduction d'une relation « *est-un* »

131

Conclusion

⇒ Utiliser l'héritage pour la traduction d'une relation « *est-un* » statique, mais utiliser la composition et la délégation dans les autres cas

132