

# Héritage, classes abstraites et interfaces

Université de Nice - Sophia Antipolis

Version 7.0 – 27/12/11

Richard Grin

## Plan de cette partie

- Héritage
- Classe **Object**
- Polymorphisme
- Classes abstraites
- Interfaces
- Réutiliser des classes
- Mécanisme de la liaison retardée

R. Grin

Java : héritage et polymorphisme

2

## Héritage

R. Grin

Java : héritage et polymorphisme

3

## Réutilisation

- Pour raccourcir les temps d'écriture et de mise au point du code d'une application, il est intéressant de pouvoir réutiliser du code déjà utilisé

R. Grin

Java : héritage et polymorphisme

4

## Réutilisation par des classes clientes

- Soit une classe **A** dont on a le code compilé
- Une classe **C** veut réutiliser la classe **A**
- Elle peut créer des instances de **A** et leur demander des services
- On dit que la classe **C** est une classe cliente de la classe **A**

R. Grin

Java : héritage et polymorphisme

5

## Réutilisation avec modifications

- On peut souhaiter modifier en partie le comportement de **A** avant de la réutiliser :
  - le comportement de **A** convient, sauf pour des détails qu'on aimerait changer
  - on aimerait ajouter une nouvelle fonctionnalité à **A**

R. Grin

Java : héritage et polymorphisme

6

## Réutilisation avec modifications du code source

- ❑ On peut copier, puis modifier le code source de **A** dans des classes **A1**, **A2**,...
- ❑ Problèmes :
  - code source de **A** pas toujours disponible
  - les améliorations futures du code de **A** ne seront pas dans les classes **A1**, **A2**,... (et réciproquement) ; difficile à maintenir !

R. Grin

Java : héritage et polymorphisme

7

## Réutilisation par l'héritage

- ❑ L'héritage existe dans tous les langages objet à classes
- ❑ 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**
- ❑ On a seulement besoin du code compilé de **A**

R. Grin

Java : héritage et polymorphisme

8

## Réutilisation par l'héritage

- ❑ Le code source de **B** ne comporte que ce qui a changé par rapport au code de **A**
- ❑ On peut par exemple
  - ajouter de nouvelles méthodes
  - modifier certaines méthodes

R. Grin

Java : héritage et polymorphisme

9

## 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

R. Grin

Java : héritage et polymorphisme

10

## Exemple d'héritage - classe mère

```
public class Rectangle {
    private int x, y; // sommet en haut à gauche
    private int largeur, hauteur;
    // La classe contient des constructeurs,
    // des 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);
    }
}
```

R. Grin

Java : héritage et polymorphisme

11

## Exemple d'héritage - classe fille

```
public class RectangleCouleur 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());
    }
}
```

R. Grin

Java : héritage et polymorphisme

12

## Code des classes filles

- ❑ Quand on écrit la classe **RectangleColore**, on doit seulement
  - écrire le code (variables ou méthodes) lié aux nouvelles possibilités ; on ajoute ainsi une variable **couleur** et les méthodes qui y sont liées
  - redéfinir certaines méthodes ; on redéfinit la méthode **dessineToi()**

R. Grin

Java : héritage et polymorphisme

13

## Exemples d'héritages

- ❑ Classe mère **Vehicule**, classes filles **veLo**, **Voiture** et **Camion**
- ❑ Classe **Avion**, classes mères **ObjetVolant** et **ObjetMotorise**
- ❑ Classe **Polygone** hérite de la classe **FigureGeometrique**
- ❑ Classe **Image**, classes filles **ImageGIF** et **ImageJpeg**

R. Grin

Java : héritage et polymorphisme

14

## 2 façons de voir l'héritage

- ❑ Particularisation-généralisation :
  - 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 de nouveaux services ou enrichit les services rendus par une classe : la classe **RectangleColore** permet de dessiner avec des couleurs et pas seulement en « noir et blanc »

R. Grin

Java : héritage et polymorphisme

15

- ❑ Chaque langage objet a ses particularités
- ❑ Par exemple, C++ et Eiffel permettent l'héritage multiple ; C# et Java ne le permettent pas
- ❑ A partir de ce point on décrit l'héritage dans le langage Java

R. Grin

Java : héritage et polymorphisme

16

## L'héritage en Java

- ❑ En Java, chaque classe a une et une seule classe mère (pas d'héritage multiple) dont elle hérite les variables et les méthodes
- ❑ Le mot clef **extends** indique la classe mère :

```
class RectangleColore extends Rectangle
```
- ❑ Par défaut (pas de **extends** dans la définition d'une classe), une classe hérite de la classe **Object** (étudiée plus loin)

R. Grin

Java : héritage et polymorphisme

17

## Exemples d'héritages

- ❑ **class Voiture extends Vehicule**
- ❑ **class Velo extends Vehicule**
- ❑ **class VTT extends Velo**
- ❑ **class Employe extends Personne**
- ❑ **class ImageGIF extends Image**
- ❑ **class PointColore extends Point**
- ❑ **class Polygone extends FigureGeometrique**

R. Grin

Java : héritage et polymorphisme

18

## Ce que peut faire une classe fille

- La classe qui hérite peut
  - ajouter des variables, des méthodes et des constructeurs
  - redéfinir des méthodes (même signature)
  - surcharger des méthodes (même nom mais pas même signature)
- Mais elle ne peut retirer aucune variable ou méthode

R. Grin

Java : héritage et polymorphisme

19

## 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 ; un poisson *est un* animal ; une voiture *est un* véhicule
- En Java, on évitera d'utiliser l'héritage pour réutiliser du code dans d'autres conditions

R. Grin

Java : héritage et polymorphisme

20

## Types en Java

- Le type d'une variable détermine les données que la variable peut contenir/référencer
- Le type d'une expression décrit la forme du résultat du calcul de l'expression
- Par exemple, si **x** et **y** sont des **int**, **x + y** est de type **int**
- Les types en Java : types primitifs, tableaux, énumérations et classes
- On verra aussi les interfaces et les types génériques

R. Grin

Java : héritage et polymorphisme

21

## Sous-type

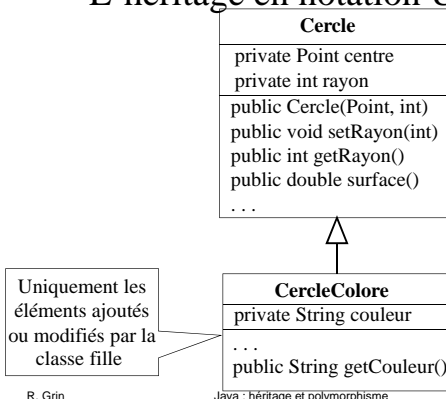
- **B** est un sous-type de **A** si on peut ranger une expression de type **B** dans une variable de type **A**
- Les sous-classes d'une classe **A** sont des sous-types de **A**
- En effet, si **B** hérite de **A**, tout **B** est un **A** donc on peut ranger un **B** dans une variable de type **A**
- Par exemple,  
**A a = new B(...);**  
est autorisé

R. Grin

Java : héritage et polymorphisme

22

## L'héritage en notation UML



R. Grin

Java : héritage et polymorphisme

23

## Compléments sur les constructeurs d'une classe

## 1ère instruction d'un constructeur

- La première instruction d'un constructeur peut être un appel
  - à un constructeur de la classe mère :  
**super(...)**
  - ou à un autre constructeur de la classe :  
**this(...)**
- Interdit de placer **this()** ou **super()** ailleurs qu'en première instruction d'un constructeur

R. Grin

Java : héritage et polymorphisme

25

## 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;
    }
    . . .
}
```

R. Grin

Java : héritage et polymorphisme

26

## Constructeurs de la classe fille

```
public class RectangleCouleur extends Rectangle {
    private Color couleur;
    public RectangleCouleur(int x, int y,
                            int largeur, int hauteur
                            Color couleur) {
        super(x, y, largeur, hauteur);
        this.couleur = couleur;
    }
    public RectangleCouleur(int x, int y,
                            int largeur, int hauteur) {
        this(x, y, largeur, hauteur, Color.black);
    }
    . . .
}
```

R. Grin

Java : héritage et polymorphisme

27

## 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 un appel implicite **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

R. Grin

Java : héritage et polymorphisme

28

## Toute première instruction exécutée par un constructeur

- Mais 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...
- Donc la toute, toute première instruction qui est exécutée par un constructeur est le constructeur (sans paramètre) de la classe **Object** !
- (C'est le seul qui sait comment créer un nouvel objet en mémoire)

R. Grin

Java : héritage et polymorphisme

29

## 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

R. Grin

Java : héritage et polymorphisme

30

## Question...

```
class A {  
    private int i;  
    A(int i) {  
        this.i = i;  
    }  
}
```

Compile ?  
S'exécute ?

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

## Exemples de constructeurs (1)

```
import java.awt.*; // pour classe Point  
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;  
    }  
}
```

Plus de constructeur sans  
paramètre par défaut !

Appel implicite du  
constructeur Object()

## Exemples de constructeurs (2)

```
// Méthodes  
public double surface() {  
    return PI * rayon * rayon;  
}  
public Point getCentre() {  
    return centre;  
}  
public static void main(String[] args) {  
    Point p = new Point(1, 2);  
    Cercle c = new Cercle(p, 5);  
    System.out.println("Surface du cercle:"  
        + c.surface());  
}
```

## Exemples de constructeurs (3)

```
public class CercleCouleur extends Cercle {  
    private String couleur;  
    public CercleCouleur(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 ?

## Une erreur de débutant !

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

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

## Héritage – problème d'accès

```
public class Animal {  
    String nom; // pas private ; à suivre...  
    public Animal() {  
    }  
    public Animal(String unNom) {  
        nom = unNom;  
    }  
    public void setNom(String unNom) {  
        nom = unNom;  
    }  
    public String toString() {  
        return "Animal " + nom;  
    }  
}
```

## Héritage – problème d'accès

```
public class Poisson extends Animal {
    private int profondeurMax;
    public Poisson(String nom, int uneProfondeur) {
        this.nom = nom; // Et si nom est private ?
        profondeurMax = uneProfondeur;
    }
    public void setProfondeurMax(int uneProfondeur) {
        profondeurMax = uneProfondeur;
    }
    public String toString() {
        return "Poisson " + nom + " ; plonge jusqu'à "
            + profondeurMax + " mètres";
    }
}
```

R. Grin

Java : héritage et polymorphisme

37

## Résoudre un problème d'accès

```
public class Poisson extends Animal {
    private int profondeurMax;
    public Poisson(String unNom, int uneProfondeur) {
        super(unNom); // convient même si nom est private
        profondeurMax = uneProfondeur;
    }
    public void setProfondeurMax(int uneProfondeur) {
        profondeurMax = uneProfondeur;
    }
    public String toString() {
        return "Poisson " + getNom()
            + " plonge jusqu'à " + profondeurMax
            + " mètres";
    }
}
```

Accesneur obligatoire  
si nom est private

R. Grin

Java : héritage et polymorphisme

38

## this et constructeurs

- ❑ **this** existe dès que l'objet a été créé par le constructeur de la classe **Object**
- ❑ **this** n'existe pas avant, dans la remontée vers le constructeur de la classe **Object**
- ❑ Pratiquement, **this** existe au retour du premier appel à **super()** ou à **this()**, mais pas avant
- ❑ Ainsi **this** ne peut être utilisé (explicitement ou implicitement) dans les paramètres de **super()** ou de **this()**
- ❑ On ne peut donc pas faire un appel à une méthode d'instance dans les arguments passés à **super()** ou à **this()**

R. Grin

Java : héritage et polymorphisme

39

## this et constructeurs

- ❑ Durant la redescende dans les constructeurs des classes ancêtres, le type de l'objet en cours de création (**this**) est son type réel
- ❑ C'est utile de le savoir s'il y a un appel polymorphe dans un des constructeurs (à éviter !)

R. Grin

Java : héritage et polymorphisme

40

## Appel d'une méthode d'instance en argument d'un constructeur

- ❑ Si **traitement** est une méthode d'instance de la classe **Classe**, le code **new Classe(traitement(...), ...)** est interdit depuis une autre classe
- ❑ Exemple (**calculePrix** est une méthode d'instance de la classe **Lot**):  
**new Lot(calculePrix(article), article);**
- ❑ 2 manières interdites de résoudre le problème :
  - **int v = traitement(...);**  
**this(v, ...); // ou super(v, ...);**
  - **this(traitement(...), ...);**

R. Grin

Java : héritage et polymorphisme

41

## Résoudre ce problème

- ❑ Revoir la conception ; souvent la meilleure solution, mais ça n'est pas toujours possible :
  - Ne pas mettre de constructeur de **Lot** qui nécessite la donnée du prix du lot ; pas de variable prix dans **Lot** ; la méthode **getPrix()** calcule le prix quand on le demande
- ❑ Utiliser un autre constructeur et faire l'appel à la méthode d'instance dans le constructeur  
**this(article); // autre constructeur**  
**prix = calculePrix(article);**
- ❑ Utiliser une méthode **static** (pas d'utilisation de **this** implicite dans les paramètres du constructeur) si c'est un traitement qui n'est pas lié à une instance particulière

R. Grin

Java : héritage et polymorphisme

42

## Ordre d'exécution des initialisations au chargement d'une classe

1. Initialisation des variables statiques, à leur valeur par défaut, puis aux valeurs données par le programmeur
2. Exécution des blocs initialiseurs statiques

R. Grin

Java : héritage et polymorphisme

43

## Ordre d'exécution des initialisations à la création d'une instance

1. Initialisation des variables d'instance à la valeur par défaut de leur type
2. Appel du constructeur de la classe mère (explicite ou implicite) ; éviter les appels de méthodes redéfinies car les variables d'instances ne sont pas encore initialisées dans les classes filles
3. Initialisations des variables d'instances si elles sont initialisées dans leur déclaration
4. Exécution des blocs initialiseurs d'instance (étudiés plus loin)
5. Exécution du code du constructeur

R. Grin

Java : héritage et polymorphisme

44

## Accès aux membres hérités Protection **protected**

R. Grin

Java : héritage et polymorphisme

45

## De quoi hérite une classe ?

- Si une classe **B** hérite de **A** (**B extends A**), elle hérite automatiquement et implicitement de tous les membres de la classe **A**
- Cependant la classe **B** peut ne pas avoir accès à certains membres dont elle a implicitement hérité (par exemple, les membres **private**)
- Ces membres sont utilisés pour le bon fonctionnement de **B**, mais **B** ne peut pas les nommer ni les utiliser explicitement

R. Grin

Java : héritage et polymorphisme

46

## Protection **protected**

- **protected** joue sur l'accessibilité des membres (variables ou méthodes) par les classes filles
- Les classes filles de **A** ont accès aux membres **protected** de la classe **A** ; les autres classes non filles de **A** n'y ont pas accès

R. Grin

Java : héritage et polymorphisme

47

## Exemple d'utilisation de **protected**

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

*Note : il est déconseillé d'utiliser des variables **protected** (voir suite de ce cours)*

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

R. Grin

Java : héritage et polymorphisme

48

## protected et paquetage

- En plus, **protected** autorise l'utilisation par les classes du même paquetage que la classe où est défini le membre ou le constructeur

## Précision sur protected

- Soit **m** un membre déclaré dans la classe **A** et d'accès **protected** ; soit **b1** une instance de **B**, classe fille de **A**
- **b1** a accès à
  - **b1.m** (sous la forme **m** ou **this.m**)
  - **b2.m** où **b2** est une instance de **B** (la granularité de protection est la classe) ou d'une sous-classe de **B**
- Mais **b** n'a pas accès à
  - **a.m** où **a** est une instance de **A**

## Précision sur protected (2)

- Attention, **protected** joue donc sur
  - l'accessibilité par **B** du membre **m** hérité (**B** comme sous-classe de **A**)
  - mais pas sur l'accessibilité par **B** du membre **m** des instances de **A** (**B** comme cliente de **A**)

## Exemple d'utilisation de protected

```
class A {  
    . . .  
    protected int m() { . . . }  
}  
  
class B extends A {  
    . . .  
    public int m2() {  
        int i = m(); // toujours autorisé  
        A a = new A();  
        i += a.m(); // pas toujours autorisé  
        . . .  
    }  
}
```

Ça dépend de quoi ?

## Pour être tout à fait précis avec protected

- Du code de **B** peut accéder à un membre **protected** de **A** (méthode **m()** ci-dessous) dans une instance de **B** ou d'une sous-classe **C** de **B** mais pas d'une instance d'une autre classe (par exemple, de la classe est **A** ou d'une autre classe fille **D** de **A**) ; voici du code de la classe **B** :  
**A a = new A(); // A classe mère de B**  
**B b = new B();**  
**C c = new C(); // C sous-classe de B**  
**D d = new D(); // D autre classe fille de A**  
**a.m(); // interdit**  
**b.m(); // autorisé**  
**c.m(); // autorisé**  
**d.m(); // interdit**

## Toutes les protections d'accès

- Les différentes protections sont donc les suivantes (dans l'ordre croissant de protection) :
  - **public**
  - **protected**
  - **package** (protection par défaut)
  - **private**
- **protected** est donc moins restrictive que la protection par défaut !

## Protections des variables

- ❑ On sait que, sauf cas exceptionnel, les variables doivent être déclarées **private**
- ❑ On peut ajouter qu'on peut quelquefois déclarer une variable **protected**, pour autoriser la manipulation directe par les futures classes filles
- ❑ Mais il est préférable de l'éviter, car cela nuit à l'encapsulation d'une classe mère par rapport à ses classes filles
- ❑ Pas de problème avec les méthodes **protected**

R. Grin

Java : héritage et polymorphisme

55

## **protected** et constructeur

- ❑ Si un constructeur de **A** est déclaré **protected**,
  - ce constructeur peut être appelé depuis un constructeur d'une classe fille **B** par un appel à **super ( )**
  - mais **B** ne peut créer d'instance de **A** par **new A ( )**

(sauf si **B** est dans le même paquetage que **A**)

R. Grin

Java : héritage et polymorphisme

56

## Classe **Object**

R. Grin

Java : héritage et polymorphisme

57

## Classe **Object**

- ❑ En Java, la racine de l'arbre d'héritage des classes est la classe **java.lang.Object**
- ❑ La classe **Object** n'a pas de variable d'instance ni de variable de classe
- ❑ La classe **Object** fournit plusieurs méthodes qui sont héritées par toutes les classes sans exception
- ❑ Les plus couramment utilisées sont les méthodes **toString** et **equals**

R. Grin

Java : héritage et polymorphisme

58

## Classe **Object** - méthode **toString ( )**

- ❑ **public String toString ( )** renvoie une description de l'objet sous la forme d'une chaîne de caractères
- ❑ Elle est utile pendant la mise au point des programmes pour faire afficher l'état d'un objet ; la description doit donc être concise, mais précise

R. Grin

Java : héritage et polymorphisme

59

## Méthode **toString ( )** de la classe **Object**

- ❑ Elle renvoie le nom de la classe, suivie de « @ » et de la valeur de la méthode **hashCode**
- ❑ La plupart du temps (à la convenance de l'implémentation) **hashCode** renvoie la valeur hexadécimale de l'adresse mémoire de l'objet
- ❑ Pour être utile dans les nouvelles classes, la méthode **toString ( )** de la classe **Object** doit donc être redéfinie

R. Grin

Java : héritage et polymorphisme

60

## Exemple

```
public class Livre {
    ...
    @Override
    public String toString() {
        return "Livre [titre=" + titre
            + ",auteur=" + auteur
            + ",nbPages=" + nbPages
            + "];"
    }
}
```

## println et toString

- Si `p1` est un objet, `System.out.println(p1)` (ou `System.out.print(p1)`) affiche la chaîne de caractères `p1.toString()` où `toString()` est la méthode de la classe de `p1`
- Il est ainsi facile d'afficher une description des objets d'un programme pendant la mise au point du programme

## Opérateur + et toString

- `toString` est utilisée par l'opérateur `+` quand un des 2 opérandes est une `String` (concaténation de chaînes) :

```
Employe e1;
...
String s = "Employe numéro " + 1 + ": " + e1;
```

## Classe Object - equals

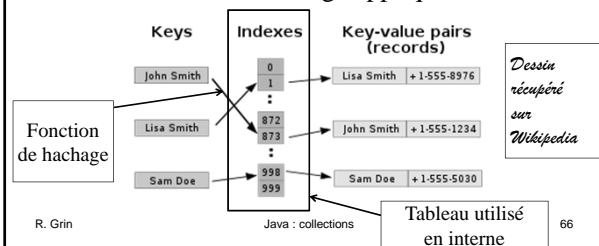
- `public boolean equals(Object obj)` renvoie `true` si et seulement si l'objet courant `this` a « la même valeur » que l'objet `obj`
- La méthode `equals` de `Object` renvoie `true` si `this` référence le même objet que `obj`
- Elle peut être redéfinie dans les classes pour lesquelles on veut une relation d'égalité (d'équivalence dirait-on en mathématiques) autre que celle qui correspond à l'identité des objets

## Problème fréquent

- Rechercher des informations en connaissant une clé qui les identifie
- Par exemple, chercher les informations sur l'employé qui a le matricule AF823
- Une table de hachage permet d'implémenter cette fonctionnalité

## Table de hachage

- Structure de données qui permet de retrouver très rapidement un objet si on connaît sa clé
- En interne l'accès aux objets utilise un tableau et une fonction de hachage appliquée à la clé



## Exemple de fonction de hachage

- ❑ Pour un nom,  
100 x longueur + la somme des valeurs des lettres
- ❑ *toto* →  $400 + 15 + 20 + 15 + 20 = 470$
- ❑ Il se peut que 2 clés différentes donnent la même valeur de hachage :  
*toto* et *otto*

R. Grin

Java : héritage et polymorphisme

67

## Clés avec une même valeur de hachage

- ❑ Techniques informatiques pour retrouver le bon objet parmi tous les objets qui ont une clé avec une même valeur de hachage, au détriment des performances
- ❑ Par exemple, ranger ces objets dans une liste chaînée qu'il faut alors parcourir pour retrouver le bon objet
- ❑ Une bonne fonction de hachage ne donne pas trop de valeurs égales pour des clés différentes

R. Grin

Java : collections

68

## hashCode

- ❑ `public int hashCode()`  
méthode de la classe `Object` qui est utilisée comme fonction de hachage par les tables de hachage fournies par le JDK
- ❑ La spécification de Java précise que 2 objets égaux au sens de `equals` doivent renvoyer le même entier pour `hashCode`
- ❑ Toute classe qui redéfinit `equals` doit donc redéfinir `hashCode`

R. Grin

Java : héritage et polymorphisme

69

## Écriture de hashCode

- ❑ La valeur calculée doit
  - ne pas être trop longue à calculer
  - ne pas avoir trop de valeurs calculées égales, pour ne pas nuire aux performances des tables de hachage qui utilisent ces valeurs
  - renvoyer la même valeur pour 2 objets qui sont égaux au sens de `equals`
- ❑ Il n'y a pas de formule universelle et il est souvent difficile de trouver une bonne méthode `hashCode`

R. Grin

Java : héritage et polymorphisme

70

## Valeurs significatives pour equals

- ❑ Il faut commencer par extraire de l'état des objets des valeurs significatives pour la méthode `equals`
- ❑ Ce sont des valeurs telles que, 2 objets sont égaux au sens de `equals` si et seulement s'ils ont les mêmes valeurs
- ❑ Le plus souvent ce sont les valeurs de certains champs des objets (par exemple le nom et le prénom)

R. Grin

Java : héritage et polymorphisme

71

## Valeurs significatives pour equals

- ❑ Quelquefois il faut transformer les champs des objets pour obtenir ces valeurs significatives
- ❑ Ainsi, dans l'exemple à suivre des fractions, il faut réduire les fractions

R. Grin

Java : héritage et polymorphisme

72

## Exemple de hashCode

- Il faut alors trouver à partir des valeurs significatives une formule qui donne le plus de valeurs distinctes de hashCode pour des objets distincts
- Voici une recette (citée par le livre « Effective Java ») qui montre comment combiner des valeurs significatives pour obtenir un hashCode

R. Grin

Java : héritage et polymorphisme

73

## Exemple de hashCode

- Types possibles des valeurs significatives :
  - type primitif
  - tableau
  - classe
- Si le type est une classe qui a une méthode hashCode (on suppose qu'elle est correcte), utiliser la valeur renvoyée par cette méthode
- Les transparents suivants expliquent quelles valeurs prendre pour les types primitifs et les tableaux

R. Grin

Java : héritage et polymorphisme

74

## Exemple de hashCode - types primitifs

- hashCode pour les champs *f* de type primitif :
  - `boolean` : `f ? 0 : 1`
  - `byte`, `char`, `short`, `int` : `(int)f`
  - `long` : `(int)(f ^ (f >>> 32))`
  - `float` : `Float.floatToIntBits(f)`
  - `double` : `Double.doubleToLongBits(f)` et appliquer le calcul sur le type `long`

R. Grin

Java : héritage et polymorphisme

75

## Exemple de hashCode - tableau

- Si le type du champ significatif est un tableau, combiner les éléments significatifs du tableau comme s'ils étaient des champs séparés d'un objet référencé : commencer avec la valeur 17 puis combiner itérativement les hashCodes des éléments par `31 * resultat + <code élément>`
- Si tous les éléments du tableau sont significatifs, le plus simple est d'utiliser la méthode `static java.util.Arrays.hashCode(tableau)` ajoutée par Java 5

R. Grin

Java : héritage et polymorphisme

76

## Exemple de hashCode – combiner les valeurs

- Il reste à combiner itérativement les valeurs obtenues pour toutes les valeurs significatives par la formule « magique » (voir livre « Effective Java » pour explications) suivante :
  - commencer par initialiser la variable `resultat` à 17 (la variable qui contiendra le hashCode)
  - puis boucler sur toutes les valeurs trouvées pour chacun des champs significatifs, en calculant `31 * resultat + <valeur>`

R. Grin

Java : héritage et polymorphisme

77

## Exemple de equals et toString

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

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

R. Grin

78

## Exemple de hashCode

```
/* Réduit la fraction et applique la
recette */
@Override public int hashCode() {
    Fraction f = reduire();
    return 31 * (31 * 17 + f.num) + f.den;
}
```

R. Grin

Java : héritage et polymorphisme

79

## Pour calculer hashCode...

```
private static int pgcd(int i1, int i2) {
    if(i2 == 0) return i1;
    else return pgcd(i2, i1 % i2);
}

public Fraction reduire() {
    int d = pgcd(num, den);
    return new Fraction(num/d, den/d);
}
```

R. Grin

Java : héritage et polymorphisme

80

## Autre recette pour hashCode

- Une recette simplissime qui peut convenir s'il les performances ne sont pas critiques pour la recherche dans la table de hachage (moins bonne répartition des valeurs) : transformer tous les attributs en **String**, concaténer et appliquer la méthode **hashCode** de la classe **String**

R. Grin

Java : héritage et polymorphisme

81

## Méthode getClass - classe java.lang.Class

- **public Class getClass()**  
renvoie la classe de l'objet (le type retour est un peu plus complexe comme on le verra dans le cours sur la généricité)
- Une instance de la classe **Class** représente un type (classe, interface, type primitif, tableau, énumération) utilisé par l'application

R. Grin

Java : héritage et polymorphisme

82

## Quelques instances de Class

- Classe : **fr.unice.employe.Employe.class**
- Interface : **java.util.List.class**
- Types primitifs : **int.class**,  
**boolean.class**, **double.class**
- Tableaux : **int[].class**, **Employe[].class**

R. Grin

Java : héritage et polymorphisme

83

## Classe Class – méthode getName

- La méthode **getName()** de la classe **Class** renvoie le nom complet de la classe (avec le nom du paquetage)
- La méthode **getSimpleName()** de la classe **Class** renvoie le nom terminal de la classe (sans le nom du paquetage)

R. Grin

Java : héritage et polymorphisme

84

## instanceof

- Si **x** est une instance d'une sous-classe **B** de **A**, **x instanceof A** renvoie **true**
- Pour tester si un objet **o** est de la même classe que l'objet courant, il ne faut donc pas utiliser **instanceof** mais le code suivant :

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

R. Grin

Java : héritage et polymorphisme

85

## Classe `java.util.Objects` (1/2)

- Nouvelle classe du JDK 7
- Elle fournit des méthodes de classe (**static**) utilitaires, pour les objets en général
- Quelquefois juste une couche mince pour faciliter la manipulation des valeur **null** : **compare**, **equals**, **toString**, **hashCode**, **requireNonNull** ; par exemple, `<T> int compare(T, T)` retourne 0 si les 2 objets sont **null**

R. Grin

Java : héritage et polymorphisme

86

## Classe `java.util.Objects` (2/2)

- `<T> T requireNonNull(T)` retourne l'objet passé en paramètre s'il n'est pas **null** et lance une **NullPointerException** sinon ; peut servir à affecter un paramètre à une variable tout en testant qu'il n'est pas **null**
- `hash(Object...)` génère un code de hachage pour les objets passés en paramètres ; utile pour générer le code de hachage d'un objet en se basant sur les codes de hachage des champs qui le compose ; utilise **Arrays.hashCode**

R. Grin

Java : héritage et polymorphisme

87

## Compléments sur la redéfinition d'une méthode

R. Grin

Java : héritage et polymorphisme

88

## Annotation pour la redéfinition

- Depuis Java 5 on peut annoter par **@Override** une méthode qui redéfinit une méthode d'une classe ancêtre, y compris une méthode abstraite (ou qui implémente une méthode d'une interface depuis Java 6)

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

- Très utile pour repérer des fautes de frappe dans le nom de la méthode : le compilateur envoie un message d'erreur si la méthode ne redéfinit aucune méthode

R. Grin

Java : héritage et polymorphisme

89

## Redéfinition et surcharge

- Ne pas confondre redéfinition et surcharge des méthodes
- Une méthode redéfinit une méthode héritée quand elle a la même signature que l'autre méthode
- Une méthode surcharge une méthode (héritée ou définie dans la même classe) quand elle a le même nom, mais pas la même signature, que l'autre méthode

R. Grin

Java : héritage et polymorphisme

90

## 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;
    }
}
```

Peut-on enlever  
cette instruction `if` ?

R. Grin

Java : héritage et polymorphisme

91

## 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  
(explications à la  
fin de cette partie  
du cours)

R. Grin

Java : héritage et polymorphisme

92

## `super .`

- Soit une classe `B` qui hérite d'une classe `A`
- Dans une méthode d'instance `m` de `B`,  
« `super .` » sert à désigner un membre de `A`
- En particulier, `super.m(...)` désigne la  
méthode `m` de `A` qui est en train d'être  
redéfinie dans `B` :

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

R. Grin

Java : héritage et polymorphisme

93

## Compléments sur `super .`

- On ne peut trouver `super.m()` dans une  
méthode `static m()` ; une méthode `static`  
ne peut être redéfinie
- `super.i` désigne la variable cachée `i` de la classe  
mère (ne devrait jamais arriver) ; dans ce cas,  
`super.i` est équivalent à `((A)this).i`

R. Grin

Java : héritage et polymorphisme

94

## Limite pour désigner une méthode redéfinie

- On ne peut remonter plus haut que la classe  
mère pour récupérer une méthode redéfinie :
  - pas de `cast` « `(ClasseAncetre)m()` »
  - pas de « `super.super.m()` »

R. Grin

Java : héritage et polymorphisme

95

## Polymorphisme

## Une question...

- **B** hérite de **A**  
**B** redéfinit une méthode **m()** de **A**
- Quelle méthode **m** est exécutée, celle de **A** ou celle de **B** ?

```
A a = new B(5);  
a.m();
```

**a** est une variable qui contient un objet de la classe **B** mais elle est déclarée du type **A**

- La méthode appelée ne dépend que du type réel de l'objet référencé par **a** (**B**) et pas du type déclaré de **a** (**A**). C'est

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

R. Grin

Java : héritage et polymorphisme

97

## Polymorphisme

- Le polymorphisme est le fait qu'une même écriture peut correspondre à différents appels de méthodes ; par exemple,

```
A a = x.f();  
a.m();
```

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

peut appeler la méthode **m** de **A** ou d'une sous-classe de **A** (connue seulement à l'exécution)

- Ce concept est une notion fondamentale de la programmation objet, indispensable pour une utilisation efficace de l'héritage

R. Grin

Java : héritage et polymorphisme

98

## Mécanisme du polymorphisme

- Le polymorphisme est obtenu grâce au « *late binding* » (liaison retardée) : la méthode qui sera exécutée est déterminée seulement à l'exécution, et pas dès la compilation (par le type réel de l'objet qui reçoit le message, et pas par son type déclaré)

R. Grin

Java : héritage et polymorphisme

99

## Exemple de polymorphisme

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

Méthode vide

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

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

R. Grin

Java : héritage et polymorphisme

100

## 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();  
        . . .  
    }  
}
```

C'est la méthode du type réel de **figures[i]** qui est appelée

ajoute une figure dans **figures[]**

R. Grin

Java : héritage et polymorphisme

101

## Typage statique et polymorphisme

- Le typage statique doit garantir dès la compilation l'existence de la méthode appelée : la classe déclarée de l'objet (et donc toutes les sous-classes par héritage) qui reçoit le message doit posséder cette méthode
- Ainsi, la classe **Figure** doit posséder une méthode **dessineToi()**, sinon, le compilateur refusera de compiler l'instruction **figures[i].dessineToi()**

R. Grin

Java : héritage et polymorphisme

102

## Notion importante pour la mise au point et la sécurité

- ❑ Java essaie de détecter le plus possible d'erreurs dès l'analyse statique du code source durant la compilation
- ❑ Une erreur est souvent beaucoup plus coûteuse si elle est détectée lors de l'exécution

R. Grin

Java : héritage et polymorphisme

103

## Utilisation du polymorphisme

- ❑ Bien utilisé, le 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 < figures.length; i++) {  
    if (figures[i] instanceof Rectangle) {  
        . . . // dessin d'un rectangle  
    }  
    else if (figures[i] instanceof Cercle) {  
        . . . // dessin d'un cercle  
    }  
}
```

R. Grin

Java : héritage et polymorphisme

104

## Utilisation du polymorphisme (2)

- ❑ Le polymorphisme facilite l'extension des programmes : il suffit de créer de nouvelles sous-classes sans toucher au code source déjà écrit
- ❑ Par exemple, 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) {  
    . . . // dessin d'un losange  
}
```

R. Grin

Java : héritage et polymorphisme

105

## Extensibilité

- ❑ En programmation objet, une application est dite extensible si on peut étendre ses fonctionnalités **sans toucher au code source déjà écrit**
- ❑ C'est possible en utilisant en particulier l'héritage et le polymorphisme comme on vient de le voir

R. Grin

Java : héritage et polymorphisme

106

## Mécanisme de la liaison retardée

- ❑ Soit `C` la classe réelle d'un objet `o` à qui on envoie un message « `o.m()` »
- ❑ Si le code source de la classe `C` contient la définition (ou la redéfinition) d'une méthode `m()`, c'est cette méthode qui est exécutée
- ❑ Sinon, la recherche de la méthode `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()` qui est alors exécutée

R. Grin

Java : héritage et polymorphisme

107

## Affichage et polymorphisme

- ❑ `System.out.println(objet)` affiche une description de tout objet grâce au polymorphisme
- ❑ Elle est en effet équivalente à `System.out.println(objet.toString())` et grâce au polymorphisme c'est la méthode `toString()` de la classe de `objet` qui est exécutée

R. Grin

Java : héritage et polymorphisme

108

## Types des paramètres d'une méthode et surcharge

- On sait que la méthode exécutée dépend de la classe réelle de l'objet auquel on adresse le message
- Au contraire, elle dépend des types déclarés des paramètres et pas de leur type réel (à prendre en compte en cas de méthode surchargée)

R. Grin

Java : héritage et polymorphisme

109

## Contraintes pour les déclarations des méthodes redéfinies

### Raison des contraintes

- L'héritage est la traduction de « *est-un* » : 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;**  
on doit pouvoir ranger une instance de **B** dans la variable **a**, et

*toute expression où intervient la variable **a** doit pouvoir être compilée et exécutée si **a** contient la référence à une instance de **B***

R. Grin

Java : héritage et polymorphisme

111

- Les transparents suivants étudient en toute logique les contraintes imposées aux méthodes redéfinies dans le cas d'un langage objet qui respecte le principe de base énoncé dans le transparent précédent
- On verra que Java est un peu plus restrictif que ne l'imposerait la pure logique

R. Grin

Java : héritage et polymorphisme

112

### Contraintes sur le type des paramètres et le type retour

- Soit une méthode **m** de **A** déclarée par :  
**R m(P p);**  
et redéfinie dans **B** par :  
**R' m(P' p);**
- Pour respecter le principe « *est-un* », quels types **R'** et **P'** pourrait-on déclarer pour la méthode redéfinie ?

R. Grin

Java : héritage et polymorphisme

113

### Contrainte sur le type retour

```
A a = new A();  
R r = a.m();
```

doit pouvoir fonctionner si on met dans **a** une instance de **B** :

```
A a = new B();  
R r = a.m(-);
```

**m** méthode de **B** qui renvoie une valeur de type **R'**

Quelle contrainte sur **R** et **R'** ?

- **R'** doit être affectable à **R** :
  - **R'** sous-type de **R** (on parle de covariance car **B** est aussi un sous-type de **A**)
  - ou types primitifs affectables (**short** à **int** par exemple)

R. Grin

Java : héritage et polymorphisme

114

## Contrainte sur le type des paramètres

```
A a = new A(); P p = new P();  
a.m(p);
```

doit pouvoir fonctionner si on met dans **a** une instance de **B** :

```
A a = new B(); P p = new P();  
a.m(p);
```

m méthode de B qui n'accepte que les paramètres de type P'

Quelle contrainte sur P et P' ?

- P doit être affectable à P' (pas l'inverse !):
  - P' super-type de P (contravariance)
  - ou types primitifs affectables

R. Grin

Java : héritage et polymorphisme

115

## Java et les contraintes avant JDK 5

- Java ne s'embarrassait pas de ces finesses
- Une méthode redéfinie devait avoir (par définition) exactement la même signature que la méthode qu'elle redéfinissait (sinon, c'était une surcharge et pas une redéfinition)
- Elle devait aussi avoir exactement le même type retour (sinon, le compilateur envoyait un message d'erreur)

R. Grin

Java : héritage et polymorphisme

116

## Covariance du type retour depuis le JDK 5

- Depuis le JDK 5, une méthode peut modifier d'une façon covariante, avec un sous-type (mais pas avec un type primitif affectable), le type retour de la méthode qu'elle redéfinit
- Ainsi, la méthode de la classe **Object**  
**Object clone()**  
peut être redéfinie en **C clone()**  
dans une sous-classe **C** de **Object**
- Pas de changement pour les paramètres (pas de contravariance)

R. Grin

Java : héritage et polymorphisme

117

## Java et les contraintes sur l'accessibilité

- Pour les mêmes raisons que les contraintes sur le type retour et les types des paramètres la nouvelle méthode ne doit jamais être moins accessible que la méthode redéfinie
- Par exemple, la redéfinition d'une méthode **public** ne peut être **private** mais une méthode **protected** peut être redéfinie en une méthode **public**

R. Grin

Java : héritage et polymorphisme

118

## Transtypage (*cast*)

## Vocabulaire

- Classe (ou type) réelle d'un objet : classe du constructeur qui a créé l'objet
- Type déclaré d'un objet : type donné au moment de la déclaration
  - de la variable qui contient une référence vers l'objet,
  - ou du type retour de la méthode qui a renvoyé l'objet

R. Grin

Java : héritage et polymorphisme

120

## Cast : conversions de classes

- ❑ Le « *cast* » est le fait de forcer le compilateur à considérer un objet comme étant d'un type qui n'est pas le type déclaré ou réel de l'objet
- ❑ En Java, les seuls *casts* autorisés entre classes sont les *casts* entre classe mère et classe fille
- ❑ On parle de *upcast* et de *downcast* en faisant référence au fait que la classe mère est souvent dessinée au-dessus de ses classes filles dans les diagrammes de classes

R. Grin

Java : héritage et polymorphisme

121

## Syntaxe

- ❑ Pour caster un objet *o* en classe *C* :

```
(C) o
```

- ❑ Exemple :

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

R. Grin

Java : héritage et polymorphisme

122

## UpCast : classe fille → classe mère

- ❑ *Upcast* : un objet est considéré comme une instance d'une des classes ancêtres de sa classe réelle
- ❑ Il est toujours possible de faire un *upcast* : à cause de la relation *est-un* de l'héritage, tout objet peut être considéré comme une instance d'une classe ancêtre
- ❑ Le *upcast* est souvent implicite

R. Grin

Java : héritage et polymorphisme

123

## Utilisation du *UpCast*

- ❑ Il est souvent utilisé pour profiter ensuite du polymorphisme :

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

R. Grin

Java : héritage et polymorphisme

124

## DownCast : classe mère → classe fille

- ❑ *Downcast* : un objet est considéré comme étant d'une classe fille de sa classe de déclaration
- ❑ 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
- ❑ Un *downcast* doit toujours être explicite

R. Grin

Java : héritage et polymorphisme

125

## 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*

R. Grin

Java : héritage et polymorphisme

126

## Downcast pour récupérer les éléments d'une liste (avant JDK 5)

```
// Ajoute des figures dans un ArrayList.  
// Un ArrayList contient un nombre quelconque  
// d'instances de Object  
ArrayList figures = new ArrayList();  
figures.add(new Cercle(centre, rayon));  
figures.add(new Rectangle(p1, p2));  
* * *  
// Le type retour déclaré de get() est Object. Cast  
// nécessaire car dessineToi() pas une méthode de Object  
((Figure)figures.get(i)).dessineToi();  
* * *
```

R. Grin

Java : héritage et polymorphisme

127

## cast et late binding

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

R. Grin

Java : héritage et polymorphisme

128

## Cacher une variable Cacher une méthode **static**

## Cacher une variable

- Si une variable d'instance ou de classe a le même nom qu'une variable héritée, elle définit une nouvelle variable qui n'a rien à voir avec la variable de la classe mère, et qui cache l'ancienne variable
- Il faut éviter de cacher intentionnellement une variable par une autre car cela nuit à la lisibilité

R. Grin

Java : héritage et polymorphisme

130

## Cacher une méthode **static**

- On ne *redéfinit* pas une méthode **static**, on la *cache* (comme les variables)
- Si la méthode **static** *m* de **Classe1** est cachée par une méthode *m* d'une classe fille, la différence est que
  - on peut désigner la méthode cachée de **Classe1** en préfixant par le nom de la classe : **Classe1.m()**
  - ou par un *cast* (*x* est une instance d'une classe fille de **Classe1**) : **((Classe1)x)m()**
  - mais on ne peut pas la désigner en la préfixant par « **super.** »

R. Grin

Java : héritage et polymorphisme

131

## Pas de liaison retardée avec les méthodes **static**

- La méthode qui sera exécutée est déterminée par la déclaration et pas par le type réel d'une instance
- Exemple : **VTT** est une sous-classe de **Velo**. Soit **nbVelos()** méthode **static** de **Velo** cachée par une autre méthode **static** **nbVelos()** dans **VTT**

```
Velo v1 = new VTT();  
n = v1.nbVelos();
```

```
n = Velo.nbVelos();  
(ou n = VTT.nbVelos(); )  
est plus lisible
```

C'est la méthode **nbVelo()** de la classe **Velo** qui sera exécutée

R. Grin

Java : héritage et polymorphisme

132

## Conclusion sur les méthodes **static**

- ❑ Il faut essayer d'éviter les méthodes **static** qui nuisent à l'extensibilité et ne sont pas dans l'esprit de la programmation objet
- ❑ Il existe évidemment des cas où les méthodes **static** sont utiles (voir par exemple la classe `java.lang.Math`), mais ils sont rares

R. Grin

Java : héritage et polymorphisme

133

## Compléments : final, tableaux, appel d'une méthode polymorphe dans le constructeur d'une classe mère

R. Grin

Java : héritage et polymorphisme

134

## Classe **final** (et autres **final**)

- ❑ Classe **final** : ne peut avoir de classes filles (`String` est **final**)
- ❑ Méthode **final** : ne peut être redéfinie
- ❑ Variable (locale ou d'état) **final** : la valeur ne pourra être modifiée après son initialisation
- ❑ Paramètre **final** (d'une méthode ou d'un `catch`) : la valeur (éventuellement une référence) ne pourra être modifiée dans le code de la méthode

R. Grin

Java : héritage et polymorphisme

135

## Tableaux et héritage

- ❑ Les tableaux héritent de la classe `Object`
- ❑ Si une classe `B` hérite d'une classe `A`, la classe des tableaux à 1 dimension d'instances de `B` est une sous-classe de la classe des tableaux à 1 dimension d'instances de `A` (idem pour toutes les dimensions)
- ❑ On peut donc écrire :  
`A[] tb = new B[5];`

R. Grin

Java : héritage et polymorphisme

136

- ❑ Il peut sembler naturel que `Cercle[]` soit un sous-type de `Figure[]`
- ❑ Pourtant ce fait pose des problèmes de typage

R. Grin

Java : héritage et polymorphisme

137

## Problème de typage avec l'héritage de tableaux

- ❑ Le code suivant va passer à la compilation :

```
Figure fig = new Carre(p1, p2);
Figure[] tbFig = new Cercle[5];
tbFig[0] = fig;
```
- ❑ Mais provoquera une erreur `java.lang.ArrayStoreException` à l'exécution car on veut mettre dans le tableau de `Cercle` une valeur qui n'est pas du type réel du tableau (un `Carre`)

R. Grin

Java : héritage et polymorphisme

138

## Tableaux et *cast*

```
Object[] to1 = new String[2];  
String[] ts1 = (String[])to1;
```

pas d'erreur à l'exécution car  
to1 créé avec new String[...]

```
Object[] to2 = new Object[2];  
to2[0] = "abc";  
to2[1] = "cd";  
ts1 = (String[])to2;
```

to2 ne contient que des String  
mais ClassCastException  
à l'exécution car to2 créé avec  
new Object[...]

```
Object o = (Object)ts1;
```

Un tableau peut toujours être *casté*  
en Object (*cast* implicite possible)

R. Grin

Java : héritage et polymorphisme

139

## Éviter l'appel d'une méthode polymorphe dans un constructeur

- En effet, on a vu que l'appel au constructeur de la classe mère est effectué avant que les variables d'instance ne soient initialisées dans le constructeur de la classe fille
- Si le constructeur de la classe mère comporte un appel à une méthode *m* (re)définie dans la classe fille, *m* ne pourra utiliser les bonnes valeurs pour les variables d'instance initialisées dans le constructeur de la classe fille

R. Grin

Java : héritage et polymorphisme

140

## Exemple

```
class M {  
    M() {  
        m();  
    }  
    void m() { ... }  
}
```

```
F f = new F(5);  
affichera  
i de F = 0 !
```

```
class F extends M {  
    private int i;  
  
    F(int i) {  
        super();  
        this.i = i;  
    }  
    void m() {  
        System.out.print(  
            "i de F = " + i);  
    }  
}
```

R. Grin

Java : héritage et polymorphisme

141

## Classes abstraites

R. Grin

Java : héritage et polymorphisme

142

## Méthode abstraite

- Une méthode est abstraite (modificateur **abstract**) lorsqu'on la déclare, sans donner son implémentation (pas d'accolades mais un simple « ; » à la suite de la signature de la méthode) :

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

- La méthode sera implémentée par les classes filles

R. Grin

Java : héritage et polymorphisme

143

## Classe abstraite

- Une classe doit être déclarée abstraite (**abstract class**) si elle contient une méthode abstraite
- Il est interdit de créer une instance d'une classe abstraite

R. Grin

Java : héritage et polymorphisme

144

## Compléments

- ❑ Si on veut empêcher la création d'instances d'une classe on peut la déclarer abstraite même si aucune de ses méthodes n'est abstraite
- ❑ Une méthode **static** ne peut être abstraite (car on ne peut redéfinir une méthode **static**)

R. Grin

Java : héritage et polymorphisme

145

## Exemple d'utilisation de classe abstraite

- ❑ On veut écrire un programme pour dessiner des graphiques et faire des statistiques sur les cours de la bourse
- ❑ Pour cela, le programme va récupérer les cours des actions en accédant à un site financier sur le Web
- ❑ On souhaite écrire un programme qui s'adapte facilement aux différents formats HTML des sites financiers

R. Grin

Java : héritage et polymorphisme

146

## Exemple de classe abstraite

```
public abstract class CentreInfoBourse {
    private URL[] urlsCentre;
    . . .
    abstract protected
        String lireDonnees(String[] titres);
    . . . // suite dans transparent suivant
```

- ❑ **lireDonnees** lira les informations sur les titres dont les noms sont passés en paramètre, et les renverra dans un format indépendant du site consulté

R. Grin

Java : héritage et polymorphisme

147

## Suite de la classe abstraite

```
public abstract class CentreInfoBourse {
    . . .
    public String calcule(String[] titres) {
        . . .
        donnees = lireDonnees(titres);
        // Traitement effectué sur donnees
        // indépendant du site boursier
        . . .
    }
    . . .
```

**calcule** mais n'est pas abstraite bien qu'elle utilise **lireDonnees** qui est abstraite

R. Grin

Java : héritage et polymorphisme

148

## Utilisation de la classe abstraite

```
public class LesEchos
    extends CentreInfoBourse {
    . . .
    public String lireDonnees(String[] titres) {
        // Implantation pour le site des Echos
        . . .
    }
```

- ❑ **lireDonnees** lit les données sur le site des Echos et les met sous un format standard manipulable par les autres méthodes de **CentreInfoBourse**

R. Grin

Java : héritage et polymorphisme

149

## Modèle de conception

- ❑ L'exemple précédent utilise le modèle de conception (*design pattern*) « patron de méthode » (*template method*) :
  - la classe mère définit la structure globale (le patron) d'un algorithme (méthode **calcule** de la classe **CentreInfoBourse**)
  - elle laisse aux classes filles le soin de définir des points bien précis de l'algorithme (méthode **lireDonnees**)

R. Grin

Java : héritage et polymorphisme

150

# Interfaces

R. Grin

Java : héritage et polymorphisme

151

## Définition des interfaces

- Une interface est une « classe » purement abstraite dont toutes les méthodes sont abstraites et publiques

R. Grin

Java : héritage et polymorphisme

152

## Exemples d'interfaces

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

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

public abstract  
peut être implicite

R. Grin

Java : héritage et polymorphisme

153

## Les interfaces sont implémentées par des classes

- Une classe implémente une interface **I** si elle déclare « **implements I** » dans son en-tête

R. Grin

Java : héritage et polymorphisme

154

## Classe qui implémente une interface

- `public class C implements I1 { ... }`
- 2 seuls cas possibles :
  - soit la classe **C** implémente toutes les méthodes de **I1**
  - soit la classe **C** doit être déclarée **abstract** ; Les méthodes manquantes seront implémentées par les classes filles de **C**

R. Grin

Java : héritage et polymorphisme

155

## Exemple d'implémentation

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

Exactement la même signature que dans l'interface Comparable

Les exceptions sont étudiées dans la prochaine partie du cours

R. Grin

Java : héritage et polymorphisme

156

## Implémentation de plusieurs interfaces

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

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

R. Grin

Java : héritage et polymorphisme

157

## Contenu des interfaces

- Une interface ne peut contenir que
  - des méthodes **abstract** et **public**
  - des définitions de constantes publiques (« **public static final** »)
- Les modificateurs **public**, **abstract** et **final** sont optionnels (en ce cas, ils sont implicites)
- Une interface ne peut contenir de méthodes **static**, **final**, **synchronized** ou **native**

R. Grin

Java : héritage et polymorphisme

158

## Accessibilité des interfaces

- Une interface peut avoir la même accessibilité que les classes :
  - **public** : utilisable de partout
  - sinon : utilisable seulement dans le même paquetage

R. Grin

Java : héritage et polymorphisme

159

## Les interfaces comme types de données

- Une interface peut servir à déclarer une variable, un paramètre, une valeur retour, un type de base de tableau, un *cast*,...
- Par exemple,  
**Comparable v1;**  
indique que la variable **v1** référencera des objets dont la classe implémentera l'interface **Comparable**

R. Grin

Java : héritage et polymorphisme

160

## Interfaces et typage

- Si une classe **C** implémente une interface **I**, le type **C** est un sous-type du type **I** : tout **C** peut être considéré comme un **I**
- On peut ainsi affecter une expression de type **C** à une variable de type **I**
- Les interfaces « s'héritent » : si une classe **C** implémente une interface **I**, toutes les sous-classes de **C** l'implémentent automatiquement (elles sont des sous-types de **I**)

R. Grin

Java : héritage et polymorphisme

161

## 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;
}
```

R. Grin

Java : héritage et polymorphisme

162

## instanceof

- Si un objet `o` est une instance d'une classe qui implémente une interface `Interface`,  
`o instanceof Interface` est vrai

R. Grin

Java : héritage et polymorphisme

163

## Polymorphisme et interfaces

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

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

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

R. Grin

Java : héritage et polymorphisme

164

## Polymorphisme et interfaces (suite)

```
public class Dessin {
    private Figure[] figures;
    . . .
    public void afficheToi() {
        for (int i = 0; i < nbFigures; i++)
            figures[i].dessineToi();
    }
    . . .
}
```

R. Grin

Java : héritage et polymorphisme

165

## Cast et interfaces

- On peut toujours faire des *casts* (*upcast* et *downcast*) entre une classe et une interface qu'elle implémente (et un *upcast* d'une interface vers la classe `Object`) :

```
// upcast Ville → Comparable
Comparable c1 = new Ville("Cannes", 200000);
Comparable c2 = new Ville("Nice", 500000);
. . .
if (c1.plusGrand(c2)) // upcast Comparable → Object
    // downcast Comparable → Ville
    System.out.println(((Ville)c2).nbHabitant());
```

R. Grin

Java : héritage et polymorphisme

166

## Utilisation des interfaces

- Plus on programme en Java, plus on découvre l'intérêt des interfaces
- Leurs utilisations sont très nombreuses et variées
- Les transparents suivants présentent les plus courantes

R. Grin

Java : héritage et polymorphisme

167

## A quoi servent les interfaces ?

- Garantir aux « clients » d'une classe que ses instances peuvent assurer certains services, ou qu'elles possèdent certaines propriétés (par exemple, être comparables à d'autres instances)
- Faire du polymorphisme avec des objets dont les classes n'appartiennent pas à la même hiérarchie d'héritage (l'interface joue le rôle de la classe mère, avec *upcast* et *downcast*)

R. Grin

Java : héritage et polymorphisme

168

## A quoi servent les interfaces ?

- ❑ Favoriser la réutilisation : si le type d'un paramètre d'une méthode est une interface, cette méthode peut s'appliquer à toutes les classes qui implémentent l'interface, et pas seulement à toutes les sous-classes d'une certaine classe
- ❑ Il est bon d'essayer de garder la bijection interface <--> service rendu : si une classe peut rendre plusieurs services de différentes natures, elle implémente plusieurs interfaces

R. Grin

Java : héritage et polymorphisme

169

## Les interfaces succédant des pointeurs de méthodes

- ❑ En Java il n'existe pas de pointeurs de fonctions/méthodes comme en C (voir cependant le cours sur la réflexivité)
- ❑ Une interface peut être utilisée pour représenter une méthode qui n'est connue qu'à l'exécution:
  - l'interface contient cette méthode
  - on appelle la méthode en envoyant le message correspondant à une instance de la classe implémente l'interface

R. Grin

Java : héritage et polymorphisme

170

## Les interfaces succédant des pointeurs de méthodes

- ❑ On verra l'application de ce mécanisme lors de l'étude du mécanisme écouteur-écouté dans les interfaces graphiques
- ❑ On pourra ainsi lancer une action (faire exécuter une méthode) quand l'utilisateur cliquera sur un bouton ; cette action sera représentée par une instance d'une classe qui implémente une interface « écouteur »

R. Grin

Java : héritage et polymorphisme

171

## Éviter de dépendre de classes concrètes

- ❑ Bon pour la maintenance d'une application
- ❑ Bon pour la réutilisation de vos classes

R. Grin

Java : héritage et polymorphisme

172

## Bon pour la maintenance

- ❑ Si les classes dépendent d'interfaces, il y a moins de risques de devoir les modifier pour tenir compte de modifications externes
- ❑ En effet, une interface qui représente un service « abstrait », sera sans doute moins souvent modifiée qu'une classe concrète car elle décrit une fonctionnalité et pas une implémentation particulière

R. Grin

Java : héritage et polymorphisme

173

## Bon pour la réutilisation

- ❑ De plus, une interface peut être implémentée par de nombreuses classes, ce qui rendra vos classes plus réutilisables

R. Grin

Java : héritage et polymorphisme

174

## 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

R. Grin

Java : héritage et polymorphisme

175

## Code

```
public class GUI {
    private FTP ftp;
    public GUI() {
        ftp = new FTP(...);
        ftp.setAfficheur(this);
        . . .
    }
    public void affiche(String m) {...}
}

public class FTP {
    private GUI gui;
    public void setAfficheur(GUI gui) {
        this.afficheur = gui;
    }
    . . .
    gui.affiche(message);
}
```

*Quel est le problème avec ce code?*

*Comment améliorer ?*

*La classe métier FTP ne pourra pas être utilisée avec une autre interface graphique !*

R. Grin

Java : héritage et polymorphisme

176

## Code amélioré (1)

```
public class GUI implements Afficheur {
    private FTP ftp;
    public GUI() {
        ftp = new FTP(...);
        ftp.setAfficheur(this);
        . . .
    }
    public void affiche(String m) {...}
}

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

R. Grin

Java : héritage et polymorphisme

177

## Code amélioré (2)

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

Maintenant les 2 classes GUI et FTP dépendent d'une interface plus « abstraite » qu'elles
```

Conséquences :

- **FTP** aura moins de risque de devoir être modifiée à cause d'une modification de l'afficheur de messages
- **FTP** sera plus facilement réutilisable, avec un afficheur d'une autre classe que **GUI**

R. Grin

Java : héritage et polymorphisme

178

## Un autre exemple : vérification d'orthographe

- Un vérificateur d'orthographe est représenté par une classe **Vérificateur**
- **Vérificateur** contient une méthode **verifie** qui vérifie si tous les mots d'un document sont contenus dans un dictionnaire
- **Vérificateur** est utilisé par une interface graphique représentée par la classe **GUI**

R. Grin

Java : héritage et polymorphisme

179

## Vérificateur interactif

- Si un mot n'est pas dans le dictionnaire, le vérificateur demande à l'utilisateur ce qu'il doit faire de ce mot :
  - l'ajouter dans le dictionnaire
  - l'ignorer
  - le corriger par un mot donné par l'utilisateur

R. Grin

Java : héritage et polymorphisme

180

## Code de GUI

- ❑ GUI contient une méthode `corrige(String mot)` qui affiche un mot inconnu à l'utilisateur et renvoie le choix de l'utilisateur
  - ❑ Il passe au vérificateur le texte tapé par l'utilisateur et lui demande de le vérifier :
  - ❑ Il contient ce code :
- ```
Verificateur v = new Verificateur(...);  
v.verifie(zoneTexte.getText(), this);
```
- ❑ `this` permettra au vérificateur d'appeler la méthode `corrige` pour les mots inconnus

R. Grin

Java : héritage et polymorphisme

181

## Méthode `verifie` – version 1

- ❑ 

```
void verifie(Document doc, GUI client) {  
    . . .  
    if (! dico.isCorrect(mot)) {  
        Correction corr = client.corrige(mot);  
        // Analyse l'action indiquée par  
        // le client et agit en conséquence  
        . . .  
    }
```
- ❑ Il y a un problème avec ce code

R. Grin

Java : héritage et polymorphisme

182

## Méthode `verifie` – version 1

- ❑ 

```
void verifie(Document doc, ? client) {  
    . . .  
    if (! dico.isCorrect(mot)) {  
        Correction corr = client.corrige(mot);  
        // Analyse l'action indiquée par  
        // le client et agit en conséquence  
        . . .  
    }
```

R. Grin

Java : héritage et polymorphisme

183

## Solution

- ❑ On déclare que le client implémente l'interface `Correcteur` qui contient une seule méthode `Correction corrige(String mot)`
- ❑ La méthode `verifie` a la signature :  
`Correction verifie(Document document, Correcteur correcteur);`
- ❑ Dans la classe du client on implémente la méthode `corrige`
- ❑ L'en-tête de GUI contiendra « `implements Correcteur` »

R. Grin

Java : héritage et polymorphisme

184

## Interfaces et API

- ❑ Soit une classe abstraite `Figure` d'une API
- ❑ Une classe `C` extérieure à l'API ne pourra hériter de `Figure` et d'une autre classe `A`
- ❑ Il vaut mieux ajouter dans l'API une interface `Figure` implémentée par la classe abstraite `FigureAbstraite`
- ❑ La classe `C` pourra ainsi implémenter `Figure` et hériter de l'autre classe `A`, et éventuellement réutiliser `FigureAbstraite` par délégation

R. Grin

Java : héritage et polymorphisme

185

## Interfaces et API

- ❑ Donc, pensez à ajouter des interfaces dans les API que vous écrivez

R. Grin

Java : héritage et polymorphisme

186

## Inconvénient des interfaces

- ❑ Il faut être conscient que, si une interface publique dans une API est destinée à être implémentée par des classes clientes, il sera difficile, sinon impossible, de modifier l'interface ; on ne pourra même pas ajouter des méthodes à cette interface
- ❑ En effet, l'ajout d'une méthode à l'interface rendra non compilables toutes les classes clientes qui implémentaient l'ancienne version de l'interface

R. Grin

Java : héritage et polymorphisme

187

## Inconvénient des interfaces (2)

- ❑ Une classe abstraite ne provoque pas ce problème
- ❑ On peut lui ajouter une méthode non abstraite sans casser le code des classes filles (elles héritent de cette méthode)
- ❑ Pour cette raison, bien souvent, on joint aux interfaces une classe abstraite qui implémente le maximum des méthodes de l'interface pour convenir à la plupart des sous-classes à venir

R. Grin

Java : héritage et polymorphisme

188

## Exemple de couples interface – classe abstraite

- ❑ Dans le JDK on peut trouver de nombreux exemples
- ❑ Par exemple, l'interface `List` associée à la classe abstraite `AbstractList`
- ❑ On trouve de même `TableModel` et `AbstractTableModel`

R. Grin

Java : héritage et polymorphisme

189

## Héritage d'interfaces

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

```
interface i1 extends i2, i3, i4 {  
    . . .  
}
```

- ❑ Dans ce cas, l'interface hérite de toutes les méthodes et constantes des interfaces « mères »

R. Grin

Java : héritage et polymorphisme

190

## Héritage et interface

- ❑ La notion d'héritage est relative à l'héritage de comportement mais aussi de structure
- ❑ En effet, si `B` hérite de `A`, les instances de `B` vont récupérer toutes les variables d'instance des instances de `A` (la structure de données)
- ❑ C'est souvent une mauvaise conception d'hériter d'une classe si on n'utilise pas toutes ses variables
- ❑ La notion d'interface est uniquement relative au comportement

R. Grin

Java : héritage et polymorphisme

191

## Héritage et interface

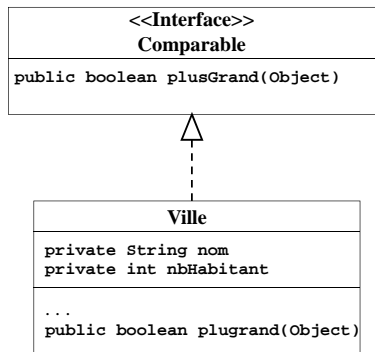
- ❑ On peut dire que lorsqu'une classe hérite d'une classe mère, elle hérite de son code et de son type
- ❑ Si elle implémente une interface, elle hérite de son type
- ❑ Hériter du code : éviter la duplication de code
- ❑ Hériter du type : polymorphisme et souplesse du sous-typage

R. Grin

Java : héritage et polymorphisme

192

## Interface en notation UML



R. Grin

Java : héritage et polymorphisme

193

## Réutilisation

R. Grin

Java : héritage et polymorphisme

194

## 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**
- Il pourrait aussi être utilisé comme moyen pratique de réutilisation de code. Par exemple, la classe **ParallélépipèdeRectangle** pourrait hériter de la classe **Rectangle** en ajoutant une variable **profondeur**

**A éviter !**

R. Grin

Java : héritage et polymorphisme

195

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

- Soit **C1** une classe déjà écrite dont on ne possède pas le code source
- 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

R. Grin

Java : héritage et polymorphisme

196

## Délégation « pure »

- Une méthode **m2()** de la classe **C2** délègue une partie de son travail à un objet **c1** de la classe **C1**, créé par la méthode **m2** :

```

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

création d'une instance de C1

utilisation de l'instance

R. Grin

Java : héritage et polymorphisme

197

## Délégation « pure » - variante

- L'objet **c1** de la classe **C1** est passé en paramètre de la méthode **m2** :

```

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

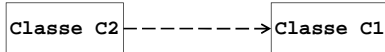
R. Grin

Java : héritage et polymorphisme

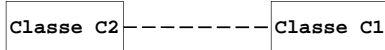
198

## En UML

- S'il y a délégation pure, la classe **C2** dépend de la classe **C1** (trait en pointillés)



- Si **C1** dépend aussi de **C2** :



R. Grin

Java : héritage et polymorphisme

199

## Délégation avec composition

- **o1** est une variable d'instance de **C2** :

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

pourra être utilisée à plusieurs occasions ; créée dans le constructeur ou ailleurs

utilisation de **o1**

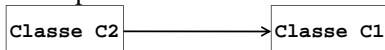
R. Grin

Java : héritage et polymorphisme

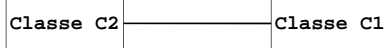
200

## En UML

- Il y a une association entre la classe **C2** et la classe **C1** (trait plein)
- L'association est unidirectionnelle si **C1** ne connaît pas **C2** :



- ou bidirectionnelle si **C1** connaît aussi **C2** :



R. Grin

Java : héritage et polymorphisme

201

## Types d'association

- Il peut y avoir plusieurs types d'association selon qu'un objet d'une classe peut être associé à un ou plusieurs objets de l'autre classe (multiplicité) ou selon le nombre de classes qui interviennent dans l'association (arité)
- Ces notions seront étudiées dans d'autres cours de conception objet

R. Grin

Java : héritage et polymorphisme

202

## Exemples de réutilisation

- Le contour d'une fenêtre dessinée sur l'écran est un rectangle
- Comment réutiliser les méthodes d'une classe **Rectangle** (comme `{get|set}Dimension()`) dans la classe **Fenetre** ?

R. Grin

Java : héritage et polymorphisme

203

## Réutilisation par héritage

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

    // Hérite de getDimension()

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

R. Grin

Java : héritage et polymorphisme

204

## Réutilisation par composition

```
public class Fenetre {
    private Rectangle contour;
    /** Renvoie la taille de la fenêtre */
    public Dimension getDimension() {
        return contour.getDimension();
    }
    /** Modifie la taille de la fenêtre */
    public void setDimension(int largeur,
                             int longueur) {
        contour.setDimension(largeur, longueur);
        ... // remplacer composants de la fenêtre
    }
    ...
}
```

R. Grin

Java : héritage et polymorphisme

205

## Réutilisation par délégation (sans composition)

```
public class Employe {
    ...
    /** Calcule le salaire */
    public double getSalaire() {
        // Délègue le calcul à un comptable
        Comptable comptable = new Comptable();
        return comptable.calculerSalaire(this);
    }
    ...
}
```

R. Grin

Java : héritage et polymorphisme

206

## Simulation de l'héritage multiple

- La composition/délégation permet de simuler l'héritage multiple (« hériter » de `Classe1` et de `Classe2` par exemple)
- On hérite d'une des classes, celle qui correspond le plus au critère « *est-un* » (`Classe1` dans l'exemple) et on utilise la délégation pour utiliser le code des autres classes (`Classe2` dans l'exemple) :

```
class Classe extends Classe1 {
    private Classe2 o2; // Classe2 à réutiliser
    // par délégation
}
```

R. Grin

Java : héritage et polymorphisme

207

## Héritage multiple avec les interfaces

- Pour pouvoir utiliser le polymorphisme avec les méthodes de `Classe2`, on peut créer une interface qui contient les méthodes de `Classe2` sur lesquelles on veut faire du polymorphisme (`m21()` et `m22()` dans cet exemple)

```
interface InterfaceClasse2 {
    public int m21();
    public Object m22();
}
```

R. Grin

Java : héritage et polymorphisme

208

## Héritage multiple avec les interfaces

- On indique alors que `Classe2`, et `Classe1` implémentent l'interface `InterfaceClasse2` :

```
class Classe extends Classe1
    implements InterfaceClasse2 {
    private Classe2 o2; // On peut « redéfinir » les méthodes
    // de classe2 ou les garder telles quelles
    public int m21() { return o2.m21() + 10; }
    public Object m22() { return o2.m22(); }
}

class Classe2 implements InterfaceClasse2 {
    ...
}
```

R. Grin

Java : héritage et polymorphisme

209

## Héritage multiple avec les interfaces

- On peut alors faire du polymorphisme sur les méthodes `m21()` et `m22()` ; par exemple :

```
InterfaceClasse2[] t =
    new InterfaceClasse2[10];
t[0] = new Classe(...);
t[1] = new Classe2(...);
...
for (int i=0; i < t.length; i++) {
    x += t[i].m21();
}
```

R. Grin

Java : héritage et polymorphisme

210

## 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 on a des variables `protected`)
- ❑ Pas possible d'hériter d'une classe `final` (comme la classe `String`)
- ❑ Pas d'héritage multiple en Java

R. Grin

Java : héritage et polymorphisme

211

## Avantages de l'héritage

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

R. Grin

Java : héritage et polymorphisme

212

## Conclusion

⇒ Il est conseillé d'utiliser l'héritage pour la traduction d'une relation « *est-un* » statique (et avec héritage de la structure), mais d'utiliser la composition et la délégation dans les autres cas

R. Grin

Java : héritage et polymorphisme

213

## Précisions sur le mécanisme de la liaison retardée (*late binding*) - complément réservé aux « initiés »

R. Grin

Java : héritage et polymorphisme

214

## Le problème :

- ❑ Voici quelques appels de méthodes :  
`truc.m(i, j);`  
`mTruc(p1).m(i, j);`  
`t[2].m(i, j);`  
`Classe.m(i, j); // méthode static`

Comment est déterminée la méthode `m` qui sera exécutée ?

R. Grin

Java : héritage et polymorphisme

215

## Cas d'un appel `static`

- ❑ Appel `static` « `Classe.m(x1, x2)` » : la détermination de la méthode statique `m` se fait uniquement à partir des déclarations du programme
- ❑ Cette détermination est faite en une seule étape par le compilateur
- ❑ Le compilateur détermine la méthode en recherchant dans `Classe` la méthode la plus spécifique (compte tenu des déclarations de `x1` et `x2`)

R. Grin

Java : héritage et polymorphisme

216

## Exemple d'appel **static**

- La classe `java.lang.Math` contient les méthodes  
`public static int min(int, int)`  
`public static long min(long, long)`

- Si le programme contient

```
int a, b;  
.  
.  
c = Math.min(a, b);
```

Le compilateur ne tient pas compte de la valeur de retour

C'est la première méthode qui sera choisie par le compilateur

- Si le programme avait déclaré « `int a; long b;` », c'est la deuxième qui aurait été choisie

R. Grin

Java : héritage et polymorphisme

217

## Cas d'un appel non **static**

- Soit le code « `objet.m(x1, x2)` » où `m` est une méthode non **static**
- Le plus souvent (mais ça ne marche pas toujours), la règle suivante permet de déterminer la méthode `m` qui sera exécutée : « `m` est déterminée par le type réel de `m` et les types déclarés de `x1` et `x2` »
- En fait, la détermination de la méthode `m` est faite en 2 étapes tout à fait distinctes :
  - étape 1, à la compilation
  - étape 2, à l'exécution

R. Grin

Java : héritage et polymorphisme

218

## Étape 1 pour déterminer la méthode à exécuter

1. Pendant la compilation, le compilateur détermine une *définition-cadre* de la méthode qui sera exécutée, en utilisant uniquement les déclarations du programme  
Le compilateur recherche dans le type déclaré de `e` la méthode la plus spécifique, de nom `m` qui a une signature qui correspond aux types déclarés des paramètres `x1` et `x2`

R. Grin

Java : héritage et polymorphisme

219

## Étape 2 pour déterminer la méthode à exécuter

2. Au moment de l'exécution, la recherche d'une méthode correspondant à la définition-cadre part de la classe réelle de l'objet qui reçoit le message et remonte vers les classes mères

R. Grin

Java : héritage et polymorphisme

220

## Ce que contient la définition-cadre à l'issue de la compilation

- Classe ou interface `T` sous laquelle se fera la recherche de la méthode durant l'exécution : le type déclaré de `objet`
- Signature de la méthode : celle de la méthode la plus spécifique de `T` qui peut convenir (selon les types déclarés de `x1` et `x2`)
- Type retour de la méthode
- Mode d'invocation

R. Grin

Java : héritage et polymorphisme

221

## Mode d'invocation de la définition-cadre de la méthode

- Le mode d'invocation de la méthode peut être :
  - non virtuel (méthode `private` ; pas de *late binding*)
  - super (appel de type « `super.m()` »)
  - interface (`objet` est déclaré du type d'une interface)
  - virtuel (tous les autres cas)

R. Grin

Java : héritage et polymorphisme

222

## Étape 2 : à l'exécution (1)

- Tout d'abord la définition-cadre doit correspondre à une méthode qui existe et qui est accessible depuis l'endroit où se situe l'appel de la méthode que l'on recherche
- Cette condition est presque toujours remplie et on peut le plus souvent l'oublier pour déterminer la méthode à exécuter
- Un cas où il faut examiner cette condition : une méthode avec la protection « paquetage » est redéfinie dans le paquetage avec la protection « public »

R. Grin

Java : héritage et polymorphisme

223

## Étape 2 : à l'exécution (2)

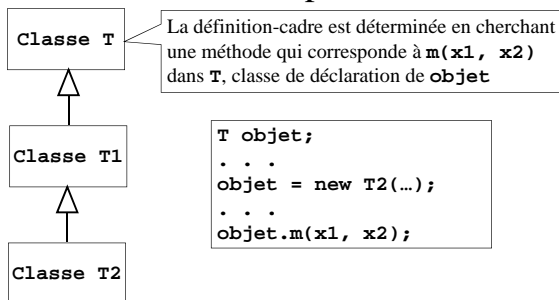
- Les actions de cette étape sont exécutées par le *bytecode* engendré par le compilateur à l'étape précédente
- 1) **objet** (à qui on envoie le message) est évalué : ça peut être **this** ou un objet quelconque
- 2) Les valeurs des arguments d'appel de la méthode sont évaluées

R. Grin

Java : héritage et polymorphisme

224

## Définition cadre durant la compilation

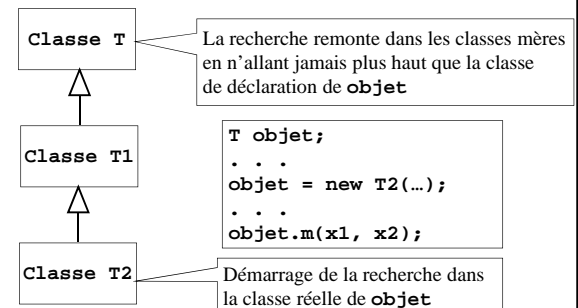


R. Grin

Java : héritage et polymorphisme

225

## Recherche de la méthode durant l'exécution



R. Grin

Java : héritage et polymorphisme

226

## Étape 2 : à l'exécution (3)

- 3) Recherche de la méthode à exécuter :
- (a) classe **C** pour démarrer la recherche : classe réelle de **objet** (ou classe mère si le mode d'invocation est « *super* »)
  - (b) si le code de la classe **C** contient une définition de méthode dont la signature est celle de la définition-cadre déterminée à la compilation, c'est la méthode que l'on cherche
  - (c) sinon, on fait la même recherche à partir de la classe mère de **C**, et ainsi de suite en remontant vers les classes mères... sans remonter au dessus de **T**, classe ou interface de base, donnée par la compilation

R. Grin

Java : héritage et polymorphisme

227

## Exécution de la méthode (détail de fonctionnement de la JVM)

- Quand la méthode a été déterminée, un cadre (*frame* en anglais) est créé, dans lequel va s'exécuter la méthode
- Ce cadre contient :
  - l'objet à qui le message est envoyé
  - les valeurs des arguments
  - l'espace nécessaire à l'exécution de la méthode (pour les variables locales, pour les appels à d'autres méthodes, ...)

R. Grin

Java : héritage et polymorphisme

228

## Le mécanisme est-il compris ?

```
class Entier {
    private int i;
    Entier(int i) { this.i = i; }
    public boolean equals(Entier e) {
        if (e == null) return false;
        return i == e.i;
    }
    public static void main(String[] args) {
        Entier e1 = new Entier(1); Entier e2 = new Entier(1);
        Object e3 = new Entier(1); Object e4 = new Entier(1);
        System.out.println(e1.equals(e2)); true ou false ? true
        System.out.println(e3.equals(e4)); true ou false ? false
        System.out.println(e1.equals(e3)); true ou false ? false !
        System.out.println(e3.equals(e1)); true ou false ? false !!
    }
}
```

ne redéfinit pas  
mais surcharge la  
méthode `equals()`  
de `Object`

R. Grin

Java : héritage et polymorphisme

229

## Une meilleure méthode `equals()`

```
public boolean equals(Object o) {
    if (! (o instanceof Entier))
        return false;
    return i == ((Entier)o).i;
}
```

redéfinit la  
méthode `equals()`  
de `Object`

□ Et il aurait encore été préférable d'écrire (pour éviter des problèmes subtils avec des éventuelles sous-classes) :

```
if (o != null && o.getClass().equals(getClass()))
    return false;
. . .
```

R. Grin

Java : héritage et polymorphisme

230