Héritage, classes abstraites et interfaces

Université Française d'Egypte
Version O 0.9 – 7/10/12
Richard Grin

Plan de cette partie

- Généralités sur l'héritage
- Classes abstraites
- Interfaces
- Réutiliser des classes

Héritage

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**

Exemple d'héritage

- Classe fille **`RectangleColore`** ; classe mère **`Rectangle`**

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
- Éviter l'héritage pour réutiliser du code dans d'autres conditions

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**
- 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**
- `A a = new B(…);` est autorisé
- `Vehicule v = new Velo();`

Compléments sur les constructeurs d'une classe

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(...)`**

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;
  }
  . . .
}
```


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);
  }
```


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 `super()` qui est un appel 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

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

Question…

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

Compile ?
S'exécute ?

Accès aux membres hérités Protection **`protected`**

Protection **`protected`**

- Membre **`m`** d'une classe **`A`** déclaré **`protected m`**
  ⇒ les classes filles de **`A`** ont accès à **`m`**
- Les classes du même paquetage y ont aussi accès
- Les autres classes n'y ont pas accès

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 !

Protections des variables

- Éviter les variables **`protected`** car elles nuisent à l'encapsulation de la classe par rapport à ses classes filles
- Pas ce problème avec les méthodes **`protected`**

Compléments sur la redéfinition d'une méthode

Annotation pour la redéfinition

- Conseillé d'annoter les méthodes qui redéfinissent une méthode

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

- Utile 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;
  }
}
```

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`**
- Quelle valeur va renvoyer cette méthode de **`B`** ?

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

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 contient une instance de **B** mais elle est déclarée du type **A**

- 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

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)

Polymorphisme

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

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() {
    . . .
}
```


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();
    . . .
```

Méthode du type réel de **figures[i]** sera exécutée

ajoute une figure dans **figures[]**

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 ?

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
  }
}
```


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
```


Extensibilité

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

Contraintes pour les déclarations des méthodes redéfinies

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`***

Contraintes « logiques » sur le type des paramètres et le type retour

- Soit une méthode **m** de **A** :
  R m(P p);
  redéfinie dans **B** par :
  R' m(P' p);
- Quelles contraintes sur les types **R'** et **P'** ?

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'**

Méthode **m** de **A** ou de **B** qui sera exécutée ?

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** (covariance car **B** est aussi un sous-type de **A**)

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)

Pas de contravariance en Java

- Java ne s'embarrasse pas de ces finesses
- Une méthode redéfinie doit avoir
  - la même signature que la méthode qu'elle redéfinit (pas de contravariance)

Covariance du type retour en Java

- Une méthode peut modifier d'une façon covariante 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**

Cast (transtypage)

Cast : conversions de classes

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

Syntaxe

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

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


- *upcast* et de *downcast* : en UML la classe mère est souvent dessinée au-dessus des classes filles

UpCast : classe fille → classe mère

- Caster vers une des classes ancêtres de la classe réelle
- Toujours possible, à cause de la relation *est-un* de l'héritage
- Souvent implicite

DownCast : classe mère → classe fille

- Caster vers 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
- Toujours explicite

Utilisation du *DownCast*

- 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 ?

Compléments sur l'héritage

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

`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 peut être modifiée après son initialisation
- Paramètre **`final`** : la valeur ne peut être modifiée dans le code de la méthode

Problème de typage avec l'héritage de tableaux

- Si **`B`** est un sous-type de **`A`**, **`B[]`** est un sous-type de **`A[]`**
- Exemple : **`Cercle[]`** est un sous-type de **`Figure[]`**
-

```
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 ?

Classes abstraites

Méthode abstraite

- Méthode sans implémentation :
  `public abstract int m(String s);`
- `m` sera implémentée par les classes filles

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

Interfaces

Définition des interfaces

- « Classe » purement abstraite dont toutes les méthodes sont abstraites et publiques

Exemples d'interfaces (1)

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


Exemples d'interfaces (1)

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

`public abstract` peut être implicite

Exemples d'interfaces (2)

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


Classe qui implémente une interface

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

Exemple d'implémentation

```
public class Ville implements Comparable {
  private String nom;
  private int nbHabitants;
  . . .
  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 ?

Exemple d'implémentation

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

Autorisé ?

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 {
  . . .
}
```


Les interfaces comme types de données

- Une interface peut servir à déclarer une variable, un type de base de tableau, un *cast*,…
- `Comparable v1;`
  les classes des objets référencés par **`v1`** devront implémenter l'interface **`Comparable`**

Interfaces et typage

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

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 ?

Polymorphisme et interfaces

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

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

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


Polymorphisme et interfaces (suite)

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


Cast et interfaces

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

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


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 les modifier pour tenir compte de modifications
  - classes plus réutilisables

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

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 ?

FTP ne pourra pas être utilisé avec une autre interface graphique !

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);
```


Code de l'interface

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