Héritage, classes abstraites et interfaces

Université de Nice - Sophia Antipolis
Version O 7.0.6 – 24/6/13
Richard Grin

Plan de cette partie

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

Héritage

Réutilisation

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

Réutilisation par des classes clientes

- Une classe **C** veut réutiliser la classe **A**
- **C** crée une instance de **A** et lui demande un service
- **C** est une classe cliente de la classe **A**

Réutilisation avec modifications

- On peut souhaiter modifier en partie le comportement de **A** avant de la réutiliser :
  - modifier une méthode/un comportement
  - ajouter une méthode/une fonctionnalité

Réutilisation avec modifications du code source

- Copier, puis modifier le code source de **A** dans des classes **A1**, **A2**,...
- Problèmes :
  - code source de **A** pas toujours disponible
  - améliorations futures du code de **A** ne seront pas dans les classes **A1**, **A2**,… Difficile à maintenir !

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**

Réutilisation par l'héritage

- Le code source de **B** ne comporte que ce qui est différent du code de **A**
- On peut par exemple
  - ajouter des méthodes
  - modifier des méthodes

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

Exemple d'héritage - classe mère

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


Exemple d'héritage - classe fille

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

Utilisation des Accesseurs public

Exemples d'héritages

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

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 des services nouveaux ou enrichis : la classe **`RectangleColore`** permet de dessiner avec des couleurs et pas seulement en « noir et blanc »

L'héritage en Java

- En Java, chaque classe a une et une seule classe mère (pas d'héritage multiple)
- **`extends`** indique la classe mère :

```
class RectangleColore extends Rectangle
```

- Par défaut, une classe hérite de la classe **`Object`**

Exemples d'héritages

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

Ce que peut faire une classe fille

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

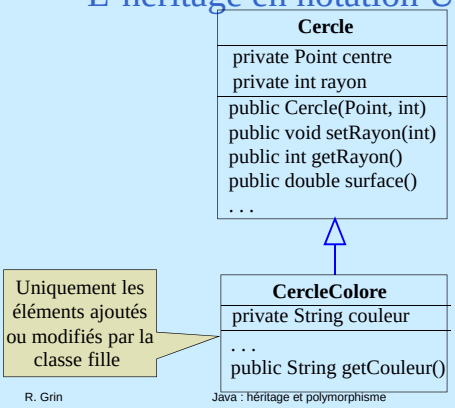
Principe important lié à la notion d'héritage

- Si « **`B extends A`** », le grand principe est que tout **`B`** est un **`A`**
- Par exemple, un rectangle coloré *est un* rectangle ; une voiture *est un* véhicule
- 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();`

L'héritage en notation UML



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 un appel **`super()`** au constructeur sans paramètre de la classe mère
  Erreur de compilation s'il n'existe pas !
- ⇒ Un constructeur de la classe mère est toujours exécuté avant les autres instructions du constructeur

Toute première instruction exécutée par un constructeur

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

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 ?

Exemples de constructeurs (1)

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

Plus de constructeur sans paramètre par défaut !

Appel implicite du constructeur `Object()`

Exemples de constructeurs (2)

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

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

Une erreur de débutant !

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

centre *et* **rayon** sont hérités de **Cercle** : ne pas les redéclarer !

Accès aux membres hérités
Protection **protected**

De quoi hérite une classe ?

- **B** hérite de **A**
- **B** hérite automatiquement et implicitement de tous les membres de la classe **A**
- Cependant **B** peut ne pas avoir accès à certains membres dont elle a hérité (par exemple, les membres **private**)

Protection **protected**

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

Exemple d'utilisation de **protected**

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

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

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


protected et paquetage

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

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 cela nuit à l'encapsulation de la classe par rapport à ses classes filles
- Pas ce problème avec les méthodes **`protected`**

Classe **`Object`**

Classe **`Object`**

- La racine de l'arbre d'héritage des classes est la classe **`java.lang.Object`**
- Pas de variable d'instance ni de variable de classe
- Les méthodes de **`Object`** sont héritées par toutes les classes
- Les plus couramment utilisées sont les méthodes **`toString`** et **`equals`**

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

- **`public String toString()`**
  renvoie une description de l'objet
- Utile pendant la mise au point ; la description doit donc être concise, mais précise
- Toute nouvelle classe devrait redéfinir sa propre méthode **`toString()`**

Classe **`Object`** - **`equals`**

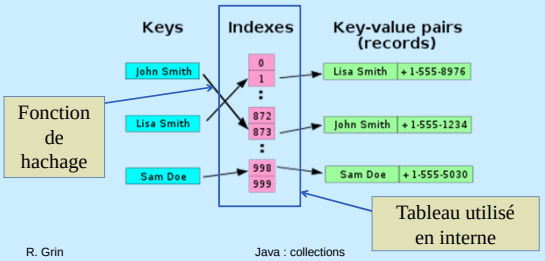
- **`public boolean equals(Object obj)`**
  renvoie **`true`**
  ssi
  `this` a « la même valeur » que **`obj`**
- La méthode **`equals`** de **`Object`** renvoie **`true`**
  ssi
  `this` référence le même objet que **`obj`**
- Peut être redéfinie dans les classes qui ont une relation d'égalité (d'équivalence)

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, avec de très bonnes performances

Table de hachage

- Structure de données qui permet de retrouver très rapidement un objet si on connaît sa 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
- 2 clés différentes peuvent donner la même valeur de hachage :
  toto et *otto*

Clés avec une même valeur de hachage

- Techniques informatiques (chaînage) pour retrouver le bon objet parmi tous les objets qui ont une clé avec une même valeur de hachage
- Impliquent de moins bonnes performances
- Bonne fonction de hachage : pas trop de valeurs de hachage égales pour des clés différentes

`hashCode`

- `public int hashCode()`
  utilisée comme fonction de hachage dans les tables de hachage du JDK
- 2 objets égaux au sens de **`equals`** doivent renvoyer le même entier pour **`hashCode`**
- Toute classe qui redéfinit **`equals`** doit donc redéfinir **`hashCode`**

Exemple de `equals` et `toString`

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

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

Exemple de **hashCode**

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


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

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

instanceof

- Si **x** est une instance d'une sous-classe **B** de **A**, **x instanceof A** renvoie **true**
- Tester si un objet **o** est de la même classe que l'objet courant :

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


Classe **java.util.Objects**

- Fournit des méthodes **static** utilitaires, pour les objets en général
- **int hash(Object…)** retourne un code de hachage pour les paramètres ; utile pour un objet en se basant sur les codes de hachage des champs qui le compose

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`**, en particulier une méthode redéfinie dans **`B`**
- 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

Polymorphisme

- Polymorphisme car une même écriture peut correspondre à plusieurs appels de méthodes
- Signature de f : `A f()`
- `A a = x.f();`
  `a.m();`
  appelle la méthode `m` de quelle classe ?

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

- Réponse : méthode m de **A** ou d'une sous-classe de **A**

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 ?

- La classe déclarée de l'objet qui reçoit le message doit posséder cette méthode

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 n'est détectée que lors de l'exécution

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
- C'est possible en utilisant l'héritage et le polymorphisme

Mécanisme de la liaison retardée

- **`o`** instance de la classe **`C`**
- Quelle classe contient la définition de la méthode exécutée par le code « **`o.m()`** » ?
- Classe **`C`** si elle contient la définition d'une méthode **`m()`**
- Sinon, la recherche de **`m()`** se poursuit dans la classe mère de **`C`**, puis dans la classe mère de cette classe mère, et ainsi de suite, jusqu'à trouver la définition d'une méthode **`m()`**

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

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`**

Contrainte sur l'accessibilité

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

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

Cast (transtypage)

Vocabulaire

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

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* : la classe mère est souvent dessinée au-dessus de ses 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

Utilisation du *UpCast*

- Pour profiter ensuite du polymorphisme :

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


Utilisation du *UpCast*

- Pour profiter ensuite du polymorphisme :

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


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*

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

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

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

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

Downcast pour récupérer les éléments d'une liste (avant Java 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.
((Figure)figures.get(i)).dessineToi();
. . .
```

Pourquoi ce *cast* ?

cast et *late binding*

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

Important !

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

Tableaux et héritage

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

Problème de typage avec l'héritage de tableaux

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

Compile ?

Et à l'exécution ?

- Erreur **`ArrayStoreException`**

Quelle ligne provoque l'erreur et pourquoi ?

Tableaux et *cast*

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

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

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

`to2` ne contient que des **`String`**

mais **`ClassCastException`** à l'exécution car **`to2`** créé avec **`new Object[..]`**

Tableaux et *cast*

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

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

Object o = (Object)ts1;
```

Un tableau peut toujours être *casté* en **`Object`**

Tableaux et *cast*

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

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

Object o = ts1;
```


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

Compléments

- Pour 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. Pourquoi ?

Exemple d'utilisation de classe abstraite

- Programme pour dessiner des graphiques et faire des statistiques sur les cours de la bourse
- Le programme récupère les cours des actions en accédant à un site financier sur le Web
- Le programme doit s'adapter facilement aux différents formats HTML des sites financiers

Exemple de classe abstraite

```
public abstract class CentreInfoBourse {
  private URL urlCentre;
  . . .
  abstract protected
    String lireDonnees(String[] titres);
  . . .
```

- **`lireDonnees`** lit les informations, et les renvoie dans un format *indépendant du site consulté*
- L'implémentation sera différente pour chaque site financier

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
    . . .
  }
```

Est-ce que **`calcule`** *est abstraite ?*

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

Utilisation de la classe abstraite

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

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);
}
```


Les interfaces sont implémentées par des classes

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

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 {
  . . .
}
```


Contenu des interfaces

- Une interface ne peut contenir que
  - des méthodes **abstract** et **public**
  - des définitions de constantes publiques (« **public static final** »)
- Les mots clés **public**, **abstract** et **final** peuvent être implicites

Accessibilité des interfaces

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

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(…);`**
- Les interfaces « s'héritent » : si une classe **`C`** implémente une interface **`I`**, toutes les sous-classes de **`C`** l'implémentent aussi

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());
```

A quoi servent les interfaces ?

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

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);
}
```

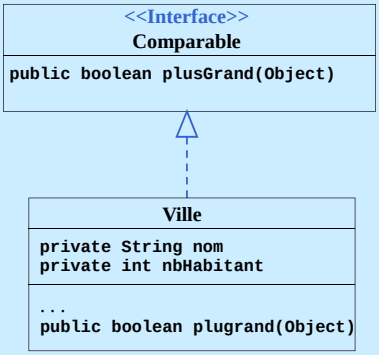
Héritage d'interfaces

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

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

- **i1** hérite de toutes les méthodes et constantes des interfaces **i2**, **i3**, **i4**

Interfaces en notation UML



Réutilisation

Règle pour l'utilisation de l'héritage

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

A éviter !

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

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

Délégation « pure »

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

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

création d'une instance de **C1**

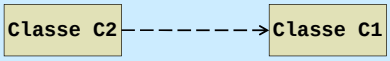
utilisation de l'instance

- Variante :

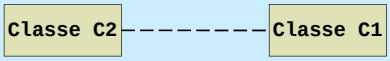
```
public int m2(C1 c1) {
  …
  r = c1.m1();
```

En UML

- La classe **C2** dépend de la classe **C1**



- Si **C1** dépend aussi de **C2** :



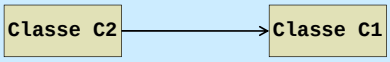
Délégation avec composition

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

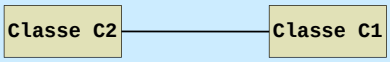
pourra être utilisé à plusieurs occasions ; créé dans le constructeur ou ailleurs

En UML

- Association entre la classe **C2** et la classe **C1**
- Unidirectionnelle si **C1** ne connaît pas **C2**:



- Bidirectionnelle si **C1** connaît **C2** :



Exemples de réutilisation

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

Réutilisation par héritage

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

  // Hérite de getDimension()

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


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 hauteur) {
    contour.setDimension(largeur, hauteur);
    ... // replacer composants de la fenêtre
  }
  . . .
}
```

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`**)

Avantages de l'héritage

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

Conclusion

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

Précisions sur la liaison retardée

Exercice

```
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);
    Object e2 = new Entier(1);
    System.out.println(e2.equals(e1));
  }
}
```

`true` ou **`false`** si méthode **`equals`** de **`Object`** ?

`true` ou **`false`** si méthode **`equals`** de **`Entier`** ?

`true` ou **`false`** ?

`false`

Position du problème

- Soit le code suivant :
  `e.m(p)`
- Quelle méthode **`m`** sera exécutée ?
- « En gros », on peut dire qu'elle est déterminée par le type réel de **`e`** et par le type déclaré de **`p`**
- En fait, c'est un peu plus complexe que cela

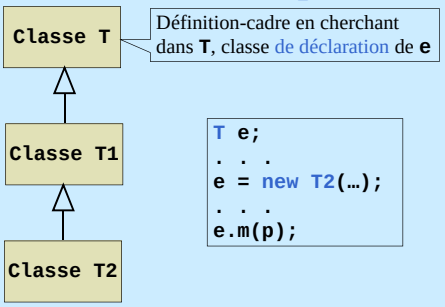
`e.m(p)` - étape 1 pour déterminer la méthode à exécuter

1. La compilation détermine une *définition-cadre* de la méthode, en utilisant uniquement les déclarations du programme :
   recherche dans le type déclaré de **`e`**
   une méthode de nom **`m`** qui a une signature qui correspond au type déclaré de **`p`**

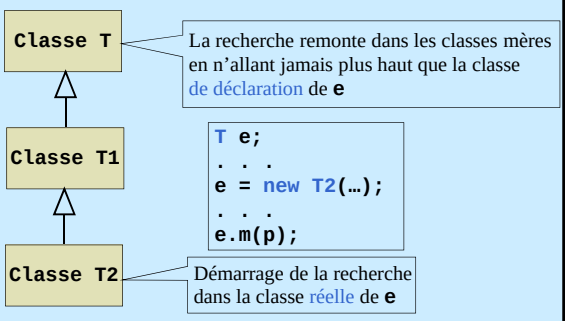
`e.m(p)` - étape 2 pour déterminer la méthode à exécuter

2. A l'exécution, recherche d'une méthode correspondant à la définition-cadre en partant de la classe réelle de **`e`** et en remontant vers les classes mères

Définition cadre durant la **compilation**



Recherche de la méthode durant **l'exécution**



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 !!
  }
}
```

surcharge la méthode **`equals()`** de **`Object`**