

# Héritage, classes abstraites et interfaces

**Université de Nice - Sophia Antipolis**

Version 6.9 – 16/1/08

Richard Grin

# Plan de cette partie

- Héritage
- Classe **Object**
- Polymorphisme
- Classes abstraites
- Interfaces
- Réutiliser des classes

# Héritage

# 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 écrit

# 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**

# Réutilisation avec modifications

- Souvent, cependant, on souhaite modifier en partie le comportement de **A** avant de le réutiliser
- Le comportement de **A** convient, sauf pour des détails qu'on aimerait changer
- Ou alors, on aimerait ajouter une nouvelle fonctionnalité à **A**

# Réutilisation avec modifications du code source

- On peut copier, puis modifier le code source de **A** dans des classes **A1**, **A2**,...
- Problèmes :
  - on n'a pas toujours le code source de **A**
  - les améliorations futures du code de **A** ne seront pas dans les classes **A1**, **A2**,...

# 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 différencessans toucher au code source de **A**
- On a seulement besoin du code compilé de **A**

# 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

# 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 en Java - classe mère

```
public class Rectangle {
    private int x, y; // point 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);
    }
}
```

# Exemple d'héritage en Java - classe fille

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

# 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()**

# Exemples d'héritages

- La classe mère **vehicule** peut avoir les classes filles **velo**, **voiture** et **Camion**
- La classe **Avion** peut avoir les classes mères **ObjetVolant** et **ObjetMotorise**
- La classe **Polygone** peut hériter de la classe **FigureGeometrique**
- La classe **Image** peut avoir comme classe fille **ImageGIF** et **ImageJpeg**

## 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 **RectangleCouleur** permet de dessiner avec des couleurs et pas seulement en « noir et blanc »

- 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

# 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)

# 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`

# 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

# 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

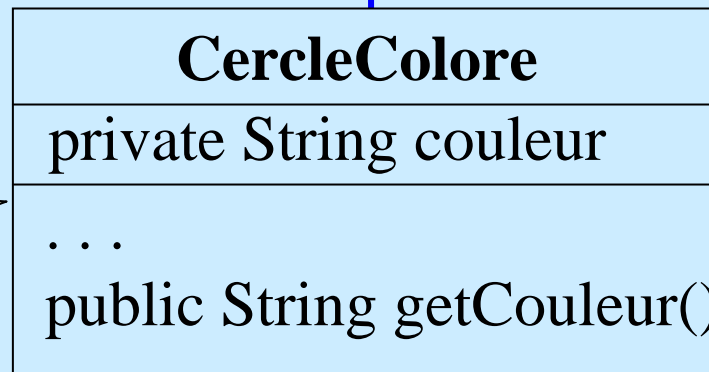
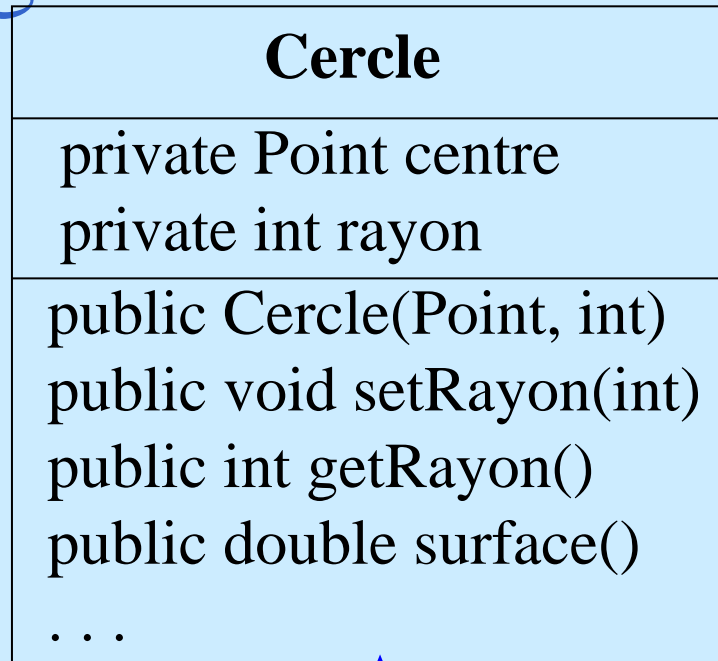
# Types en Java

- 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 et classes
- On verra aussi les interfaces et les types génériques

# 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

# L'héritage en notation UML



Uniquement les éléments ajoutés ou modifiés par la classe fille

# 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

# 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 un appel implicite **super( )** au **constructeur sans paramètre** de la classe mère (erreur 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

- 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** !
- (D'ailleurs c'est le seul qui sait comment créer un nouvel objet en mémoire)

# 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 { }
```

Ca compile ?  
Ca s'exécute ?

# Exemple 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 !

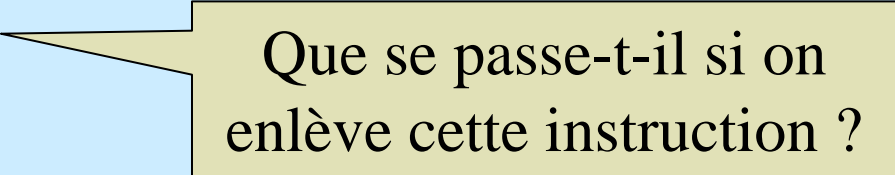
Appel implicite du constructeur Object()

# Exemple 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());
}
}
```

# Exemple de constructeurs (3)

```
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 = c;  
rayon = r;  
couleur = c;  
}
```

**centre et rayon**  
sont hérités de **Cercle** ;  
il ne faut pas les  
« redéfinir » !

# 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";
    }
}
```

# 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

# **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( )**

# 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 !)

# 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(...), ...);`

# 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

# 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

# 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) ; attention aux appels de méthodes redéfinies dans la classe : les variables d'instances ne sont pas encore initialisées
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

# Accès aux membres hérités

## Protection `protected`

# 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** (mais pas des constructeurs)
- 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

# Protection `protected`

- `protected` joue sur l'**accessibilité** des membres (variables ou méthodes) par les classes filles
- Un membre `protected` de la classe **A** peut être manipulé par les classes filles de **A** sans que les autres classes non filles de **A** ne puisse les manipuler

# 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

- 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 faut l'éviter, car cela nuit à l'encapsulation des classes mères par rapport à leurs classes filles
- Pas ce problème avec les méthodes **protected**

# 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)

# Classe Object

# 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()`

# 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

# 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

# `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 a « la même valeur » que l'objet `obj`
- La méthode `equals` de `Object` renvoie `true` si `obj` référence le même objet que `this`
- 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

# hashCode ( )

- `public int hashCode()` : méthode utilisée pour ranger les instances dans les tables de hachage
- 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`

# 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 est souvent difficile de trouver une bonne méthode `hashCode`

# Exemple de `hashCode()`

- Voici une recette (qui peut ne pas convenir) :
  1. on choisit des champs significatifs dans la classe (si la méthode **`equals`** est redéfinie, on prend un sous-ensemble des champs qui interviennent pour **`equals`**)
  2. on combine des valeurs de `hashCode` pour chacun de ces champs
- On part des `hashCode`s des types primitifs (voir transparent suivant), ou des `Strings` (fourni par la méthode **`hashCode()`**)

# Exemple de hashCode ( )

- hashCode pour **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**
  - **tableau** : combiner itérativement les hashCodes des éléments par  
**37 \* resultat + code élément**

# Exemple de `hashCode()`

- Combiner itérativement les `hashCode`s des champs choisis dans la classe par  $37 * \text{resultat} + \text{code élément}$
- Ajouter `17` au résultat et renvoyer cette valeur pour la méthode `hashCode()`
- Voir si `a.equals(b)  $\implies$  a.hashCode() == b.hashCode()`
- Si ça n'est pas le cas, modifier `hashCode...` ou choisir une autre recette !

# 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 la
   recette */
public int hashCode() {
    Fraction f = reduire();
    return (17 + f.num * 37) * 37 + f.den;
}
}
```

# 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);  
}
```

# Autres recettes pour `hashCode`

- Pour les tableaux, la classe **Arrays** fournit des méthodes **static hashCode** qui fournissent des valeurs pour les différents types de tableaux ; par exemple, **int**  
**hashCode(double[] t)**
- Une recette simplissime qui peut convenir : transformer tous les attributs en **String** et appliquer la méthode **hashCode** de la classe **String**

# Classe `Class` - méthode `getClass()`

- `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)
- Les instances de la classe `java.lang.Class` représentent les classes utilisées par Java (et aussi les types primitifs)

# 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)

# 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())
```

# Compléments sur la redéfinition d'une méthode

# Redéfinition et surcharge

- Ne pas confondre redéfinition et surcharge des méthodes :
  - on **redéfinit** une méthode quand une nouvelle méthode a la même signature qu'une méthode **héritée** de la classe mère
  - on **surcharge** une méthode quand une nouvelle méthode a le même nom, mais pas la même signature, qu'une autre méthode de la même classe

# 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;
    }
    public boolean equals(Object o) {
        if (o == null || (o.getClass() != this.getClass()))
            return false;
        return i == ((Entier)o).i;
    }
}
```

# 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)

## 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 donc en train d'être redéfinie dans **B** :

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

# 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`

# 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()** »

# Annotation pour la redéfinition

- Depuis le JDK 5 on peut accoler une annotation à une méthode qui redéfinit une méthode d'une classe ancêtre, y compris une méthode abstraite (ou qui implémente une méthode d'une interface depuis le JDK 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

# Polymorphisme

# Une question...

- Supposons que **B** hérite de **A** et que la méthode **m()** de **A** soit redéfinie dans **B**
- Quelle méthode **m()** sera exécutée dans le code suivant, celle de **A** ou celle de **B** ?

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

**a** est un objet de la classe **B**  
mais il est déclaré de la classe **A**

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

C'est **la méthode de la classe B** qui sera exécutée

# 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 de n'importe quelle sous-classe de **A**

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

# 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

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

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

ajoute les figures  
dans **figures[]**

# Typage statique et polymorphisme

- En Java, le typage statique doit garantir **dès la compilation** l'existence de la méthode appelée :

la classe **déclarée** de l'objet 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()** »

# 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

# 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  
    }  
}
```

# Utilisation du polymorphisme (2)

- Le polymorphisme **facilite l'extension** des programmes : on peut créer de nouvelles sous-classes sans toucher aux sources déjà écrits
- 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
```

# 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

# 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 de la classe **c** contient la définition (ou la redéfinition) d'une méthode **m( )**, c'est cette méthode qui sera 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

# 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

# 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)

# Contraintes pour les déclarations des méthodes redéfinies

# Raisons 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 une instance de B*

# 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 ?

# 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**)
  - ou types primitifs affectables (**short** à **int** par exemple)

# 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

# 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)

# 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)

# 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**

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 l'objet,
  - ou du type retour de la méthode qui a renvoyé l'objet

# *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

# Syntaxe

- Pour caster un objet en classe **C** :  
**(C) o;**
- Exemple :  
**Velo v = new Velo();**  
**Vehicule v2 = (Vehicule) v;**

## *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

# 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();
```

## *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** ; à l'**exécution** il sera vérifié que l'objet est bien de la classe fille
- Un *downcast* doit toujours être **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*

# *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();  
.  
.  
.
```

## *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

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

# 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.` »

# 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

# 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

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

# 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

# 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];**

# Problème de typage avec l'héritage de tableaux

- Le code suivant va passer à la compilation :

```
A a = new A();  
A[] tb = new B[5];  
tb[0] = a;
```

- Mais provoquera une erreur **ArrayStoreException** à l'exécution car on veut mettre dans le tableau une valeur qui n'est pas du type réel du tableau

# Tableaux et *cast*

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

*cast* possible 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)

# 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

# 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);  
    }  
}
```

# Classes abstraites

# 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

# Classes abstraites

- 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

# 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**)

# 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 sites financiers

# 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

# 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

# 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`

# 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**)

# Interfaces

# Définition des interfaces

- Une interface est une « classe » purement abstraite dont toutes les méthodes sont abstraites et publiques
- C'est une liste de noms de méthodes publiques

# 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

# 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** 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**

# 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

# 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 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**

# Accessibilité des interfaces

- Une interface peut avoir la 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 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**

# 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**)

# 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;  
}
```

# instanceof

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

# 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 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());
```

# 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

# 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*)

# 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  $\leftrightarrow$  service rendu : si une classe peut rendre plusieurs services de différentes natures, elle implémente plusieurs interfaces

# 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 :
  - l'interface contient cette méthode
  - on appelle la méthode en envoyant le message correspondant à une instance dont la classe implémente l'interface

# Les interfaces succédanés 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 »

# Éviter de dépendre de classes concrètes

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

# 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

# Bon pour la réutilisation

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

# Exemple typique

- Une classe « métier » **FTP** est utilisée par une interface graphique **GUI**
- **FTP** fait afficher un message d'erreur en le passant à **GUI**

# Code

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

Quel est le problème avec ce code?

Comment améliorer ?

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

La classe métier dépend de l'interface graphique !

# 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);
}
```

## 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**

# Un autre exemple : vérification d'orthographe

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

# 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

# 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

# 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

# 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  
        . . .  
    }  
}
```

# 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** »

# 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

# Interfaces et API

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

# 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, d'ajouter des méthodes à cette interface
- En effet, l'ajout d'une méthode à l'interface rendra fausses toutes les classes clientes qui implémentaient l'ancienne version de l'interface

## 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

# 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 »

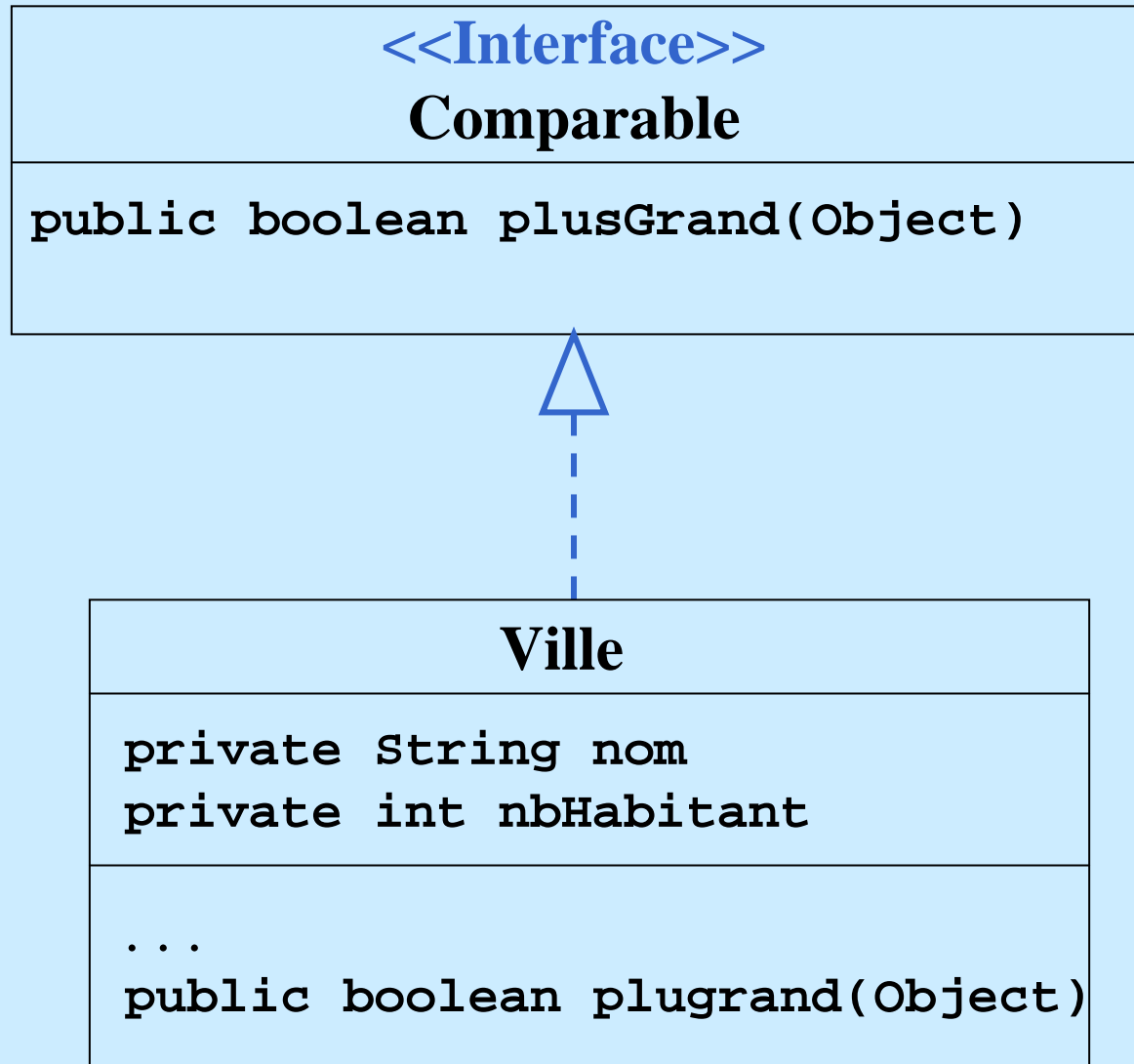
# Héritage et interface

- 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

# 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

# Interfaces en notation UML



# Réutilisation

# Utilisation de l'héritage

- Il sert à représenter la relation « *est-un* » (généralisation/spécialisation) entre classes :
  - un **CercleColore** est un **Cercle**
  - une **Voiture** est un **Vehicule**
- Mais il peut aussi être utilisé comme moyen pratique de réutilisation de code. Par exemple, la classe **ParallélépipèdeRectangle** peut hériter de la classe **Rectangle** en ajoutant une variable **profondeur**

**À éviter !**

# 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

# Délégation « pure »

- Une méthode `m2()` de la classe `C2` délègue une partie de son travail à un objet `o1` de la classe `C1`, créé par la méthode `m2()` :

```
public int m2() {  
    ...  
    C1 o1 = new C1();  
    r = o1.m1();  
    ...  
}
```

création d'une instance de C1

utilisation de l'instance

# Délégation avec composition

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

```
public class C2 {  
    private C1 o1;  
    ...  
    public int m2() {  
        ...  
        r = o1.m1();  
        ...  
    }  
}
```

utilisation de `o1`

# Exemples de réutilisation

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

# Réutilisation par héritage (1)

```
public class Fenetre extends Rectangle {  
    . . .  
    public void dessineToi() {  
        // redéfinition de la méthode : dessine  
        // une fenêtre et pas un simple rectangle  
        . . .  
    }  
}
```

## Réutilisation par héritage (2)

```
// 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
}
. . .
} // Fin classe Fenetre
```

# Réutilisation par composition

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

# 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.calculeSalaire(this);  
    }  
    . . .  
}
```

# 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
```

# 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();  
}
```

# 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
    . . .
    public int m21() { return o2.m21() + 10; }
    public Object m22() { return o2.m22(); }
}
```

On peut « redéfinir » les méthodes de **Classe2** ou les garder telles quelles

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

# 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();
}
```

# 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

# 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* »

# 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

Précisions sur le mécanisme du  
« *late binding* »  
(compléments réservés aux « initiés »)

# 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 ?

# 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`)

# 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

# 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

## 2 étapes 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**

## 2 étapes 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

# 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

# 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)

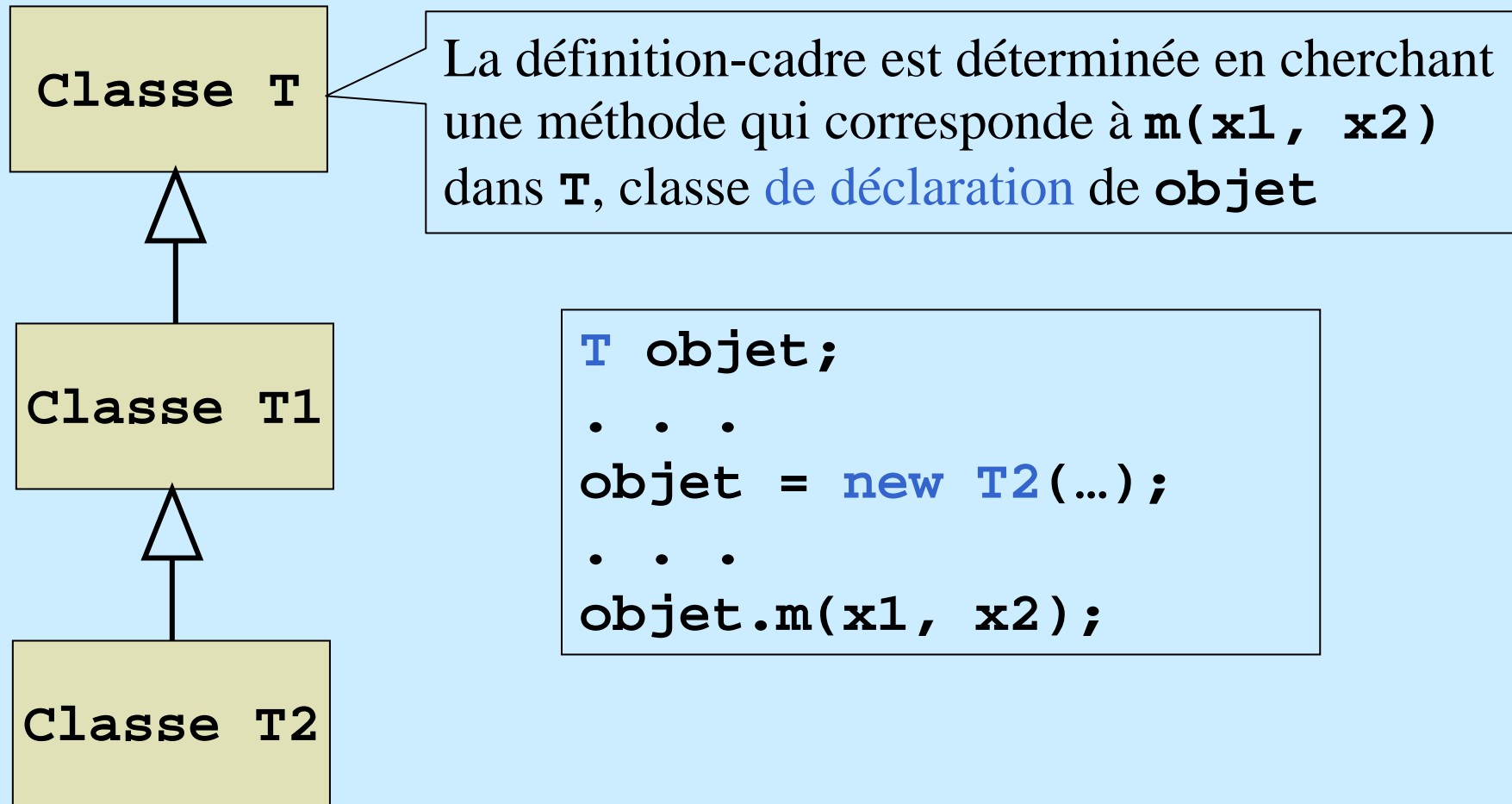
## Étape 2 : à l'exécution (1)

- 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 »

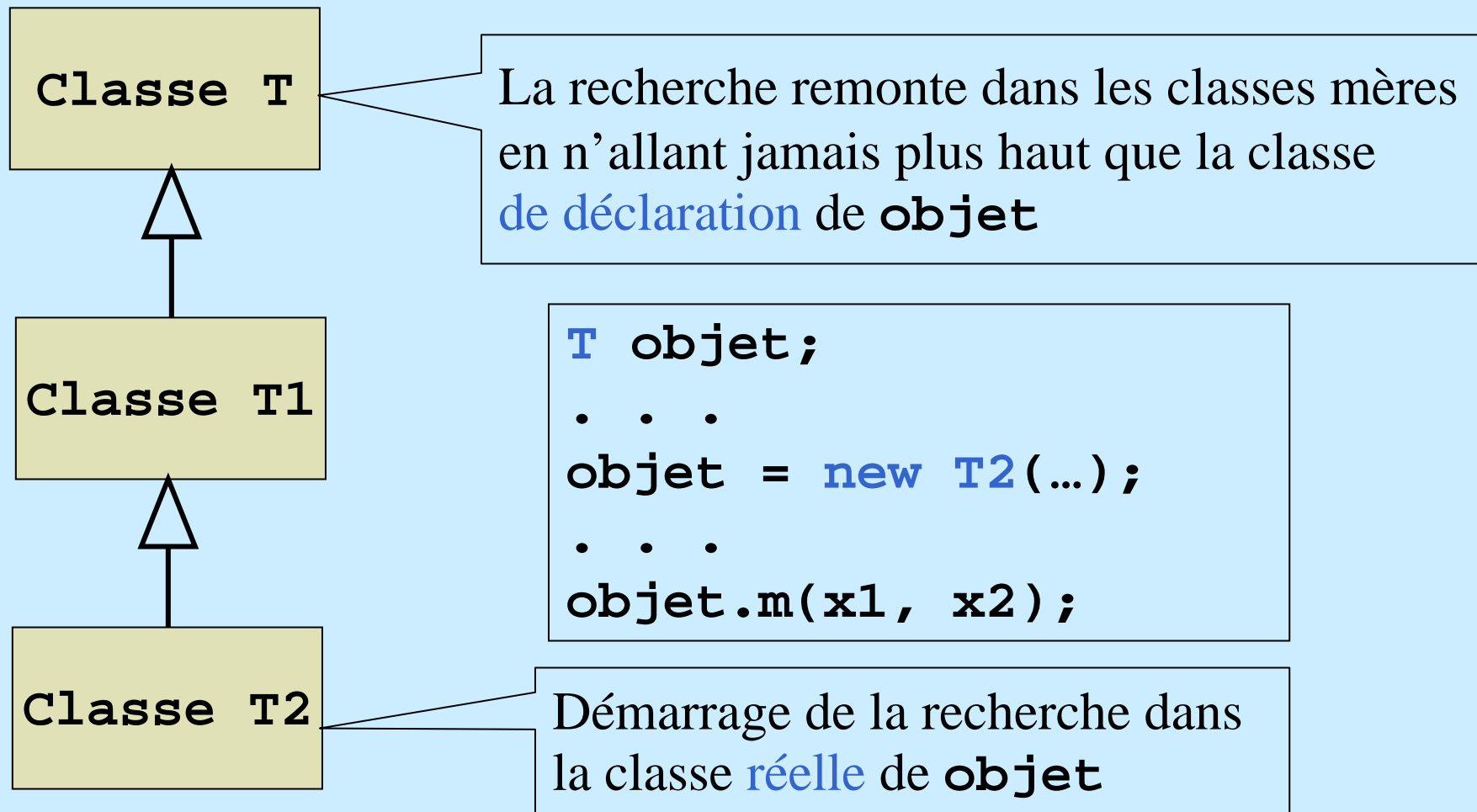
## Étape 2 : à l'exécution (2)

- 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

# Définition cadre durant la compilation



# Recherche de la méthode durant l'exécution



## 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

# 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, ...)

# 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**

# 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;  
. . .
```