Conception objet en Java avec BlueJ une approche interactive

9. Héritage

Notions avancées – polymorphisme

© David J. Barnes, Michael Kölling

Traduction et adaptation française :

© Patrice Moreaux

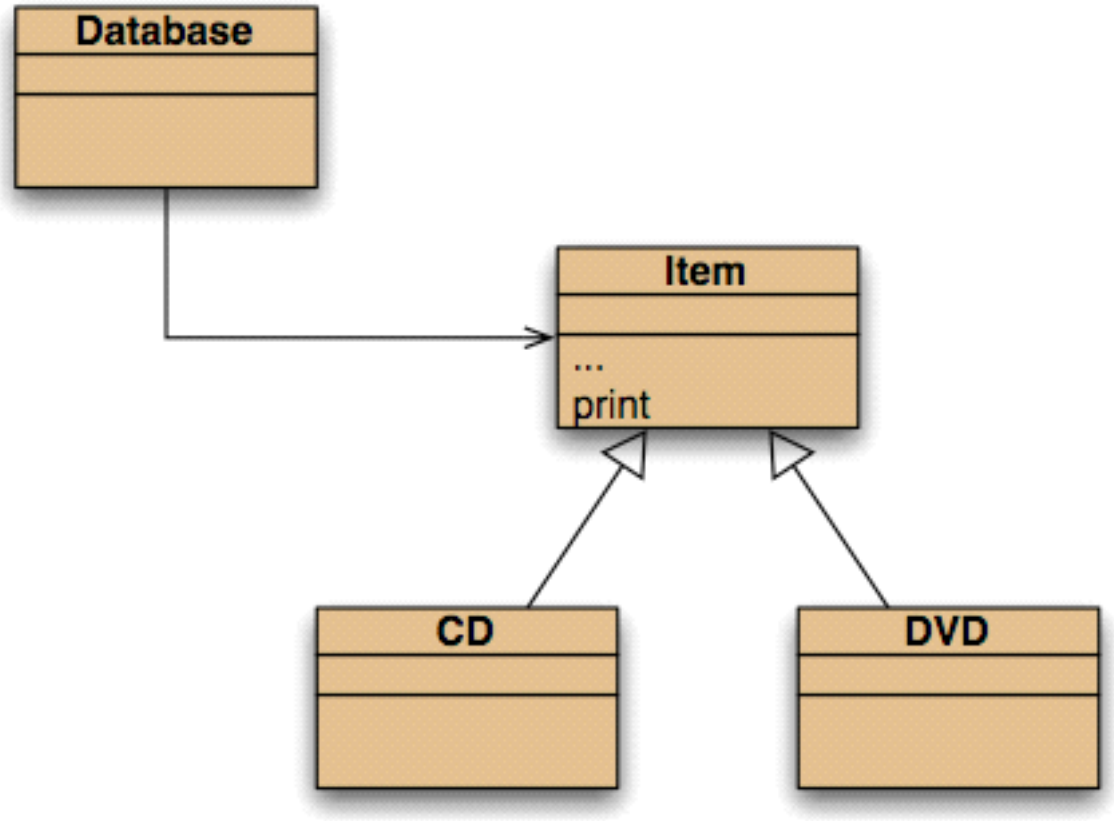
POLYTECH Annecy-Chambéry – France

Adapté pour le cours de « Programmation objet »
enseigné par Amine Brikci-Nigassa – Université de Tlemcen

Principaux concepts abordés

- Polymorphisme de méthode
- Type statique et dynamique
- Redéfinition
- Recherche dynamique de méthode
- Accès protégé

La hiérarchie d'héritage



Affichage insatisfaisant

Ce que nous voulons

```
CD: A Swingin' Affair (64 mins)*
    Frank Sinatra
    tracks: 16
    mon album préféré de Sinatra

DVD: O Brother, Where Art Thou? (106 mins)
     Joel & Ethan Coen
     Le meilleur film des frères Coen!
```

Ce que nous avons actuellement

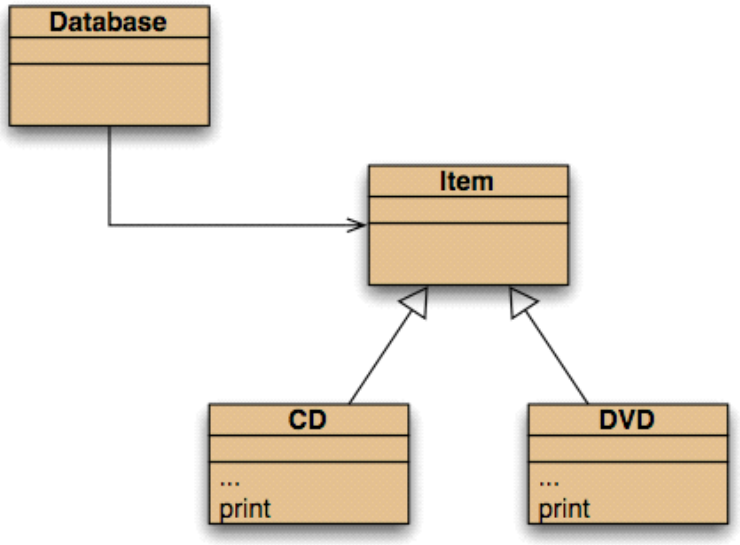
```
title: A Swingin' Affair (64 mins)*
    mon album préféré de Sinatra

title: O Brother, Where Art Thou? (106 mins)
    Le meilleur film des frères Coen!
```

Le problème

- La méthode **`print`** de **`Item`** affiche seulement les champs communs.
- L'héritage est « à sens unique » :
  - Une sous-classe hérite les champs de la superclasse.
  - La superclasse ne sait rien des champs de la sous-classe.

Une tentative de solution…



- Placer **`print`** là où elle peut accéder à l'information dont elle a besoin.
- Chaque sous-classe possède sa propre version.
- Mais les champs de **`Item`** sont privés…
- … **`Database`** ne trouve pas de méthode **`print`** dans **`Item`**.

Type statique et type dynamique (1)

- Il nous faut de nouveaux concepts pour gérer des hiérarchies de types plus complexes.
- Un peu de terminologie :
  - Type statique
  - Type dynamique
  - Sélection/recherche de méthode

Type statique et type dynamique (2)

Quel est le type de c1 ?

```
Car c1 = new Car();
```

Quel est le type de v1 ?

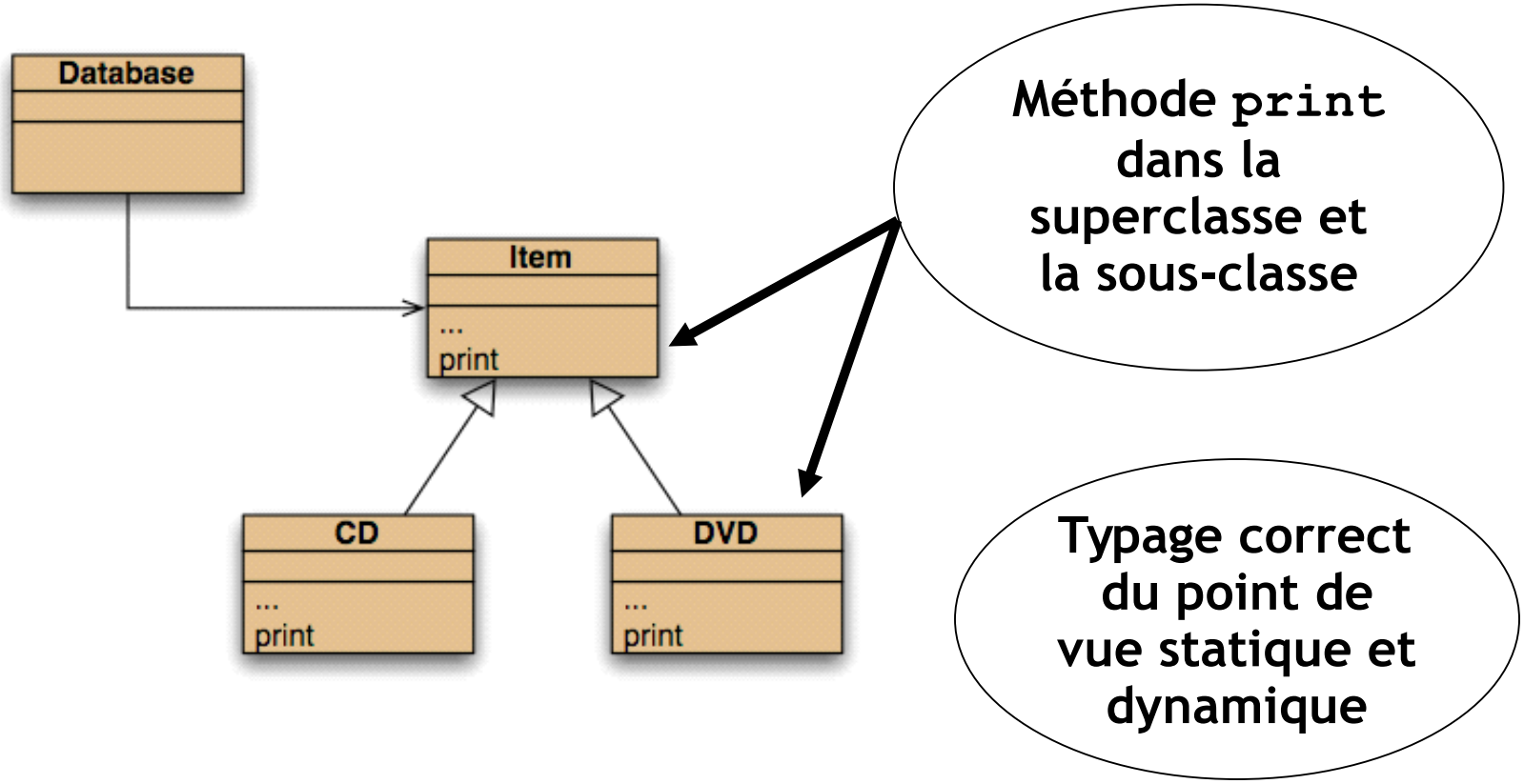
```
Vehicle v1 = new Car();
```

Type statique et type dynamique (3)

- Le type déclaré d'une variable est son *type statique*.
- Le type de l'objet que référence une variable est le *type dynamique* de cette variable.
- Le compilateur contrôle les violations de type statique.

```
for (Item item : items){
    item.print(); // erreur de compilation.
}
```

Solution : la redéfinition

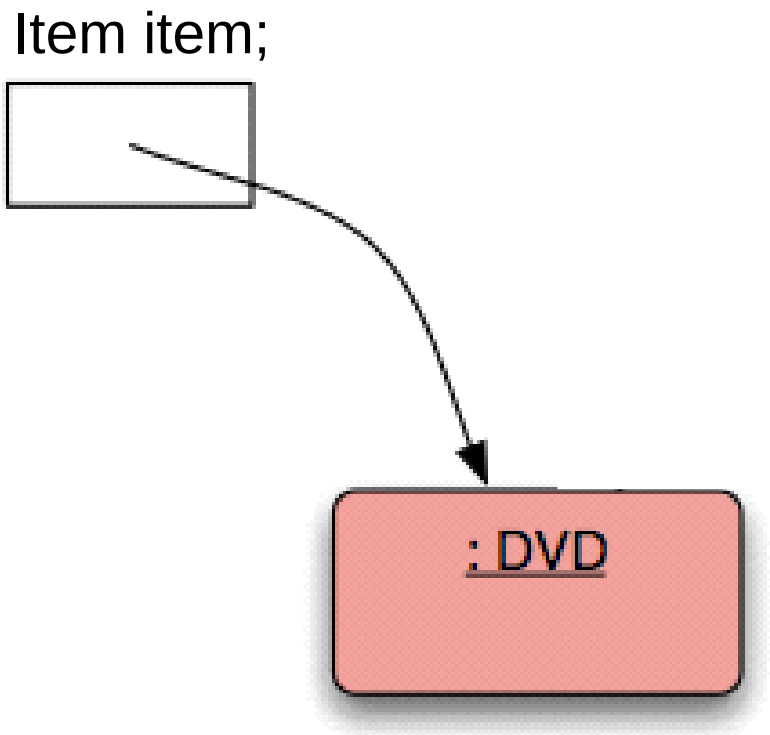


Redéfinition

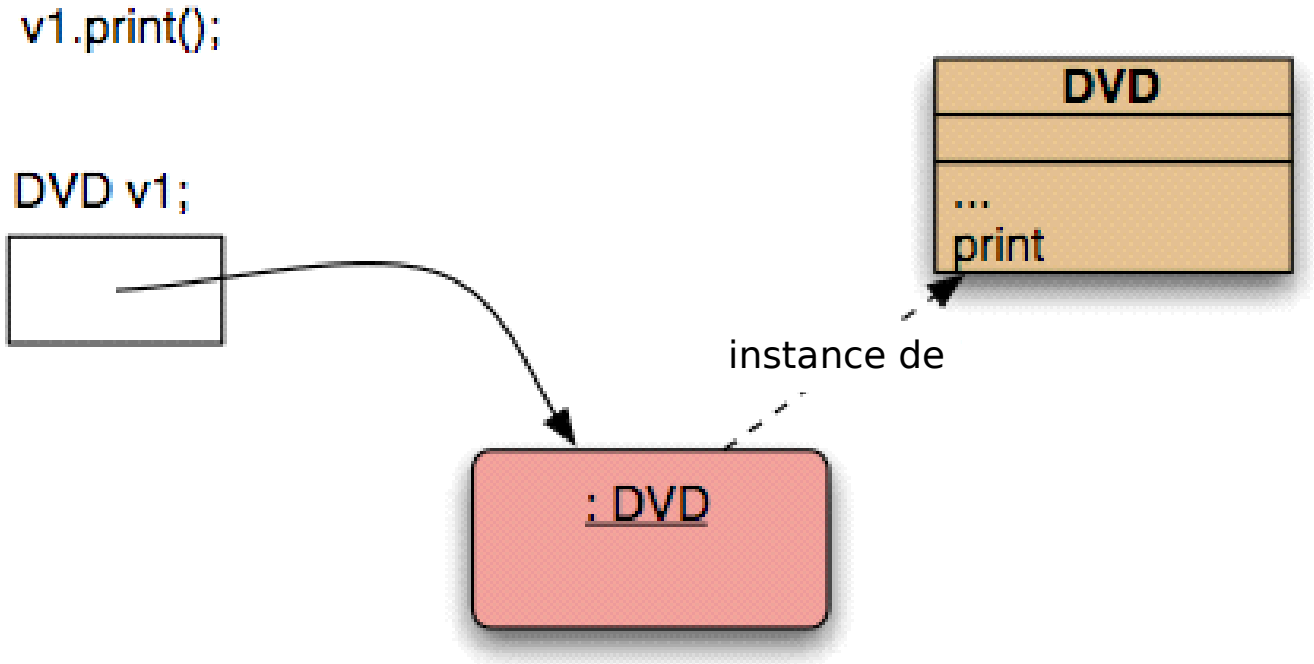
(en anglais : *overriding*)

- La superclasse et la sous-classe définissent des méthodes de même signature.
- Chacune a accès aux champs de sa classe.
- La vérification statique de type porte sur la superclasse.
- La méthode *redéfinie* dans la sous-classe est appelée à l'exécution – elle remplace la version de la superclasse.
- Que devient la version de la superclasse ?

Types statique et dynamique différents

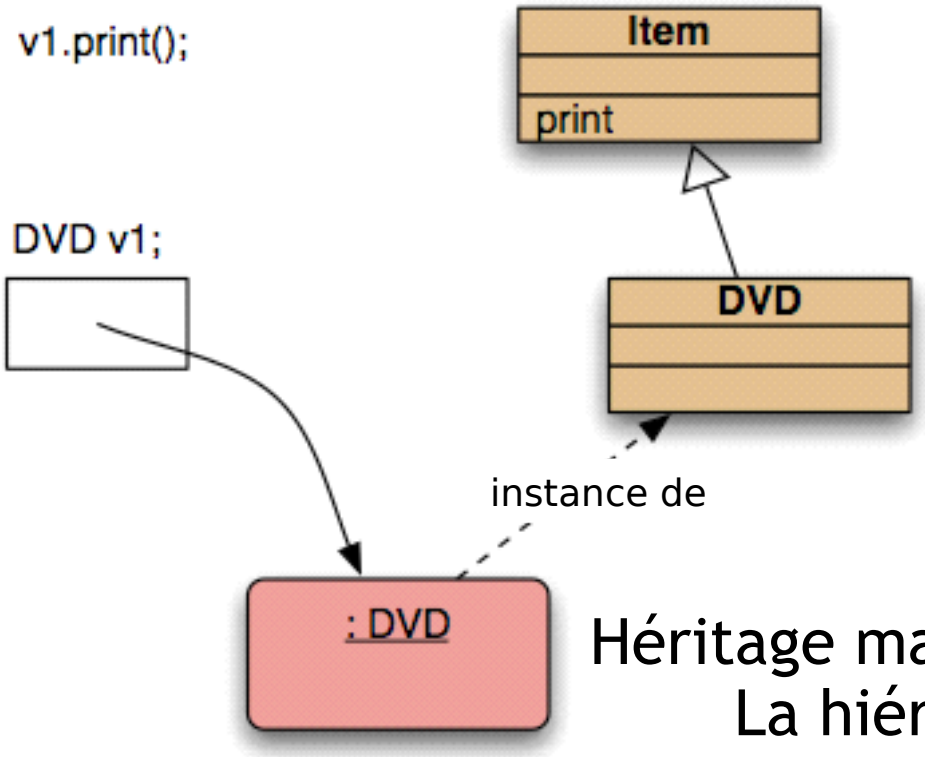


Recherche/sélection de méthode (1)



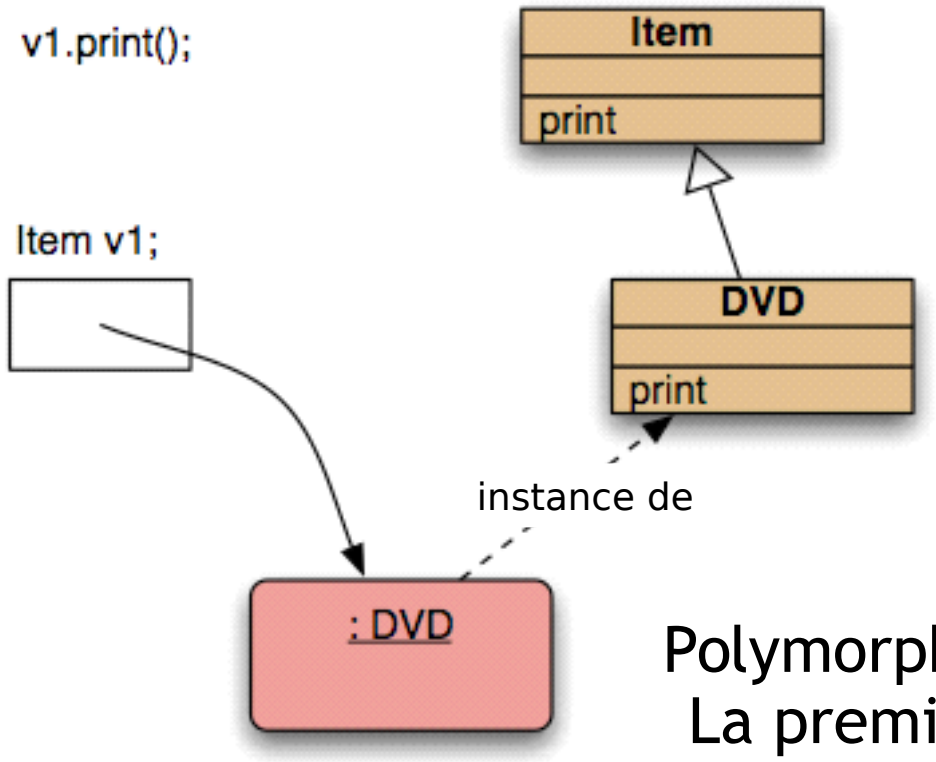
Pas d'héritage ni de polymorphisme.
La seule méthode possible est sélectionnée.

Recherche/sélection de méthode (2)



Héritage mais pas de redéfinition. La hiérarchie d'héritage est parcourue de bas en haut à la recherche de la méthode.

Recherche/sélection de méthode (3)



Polymorphisme et redéfinition.
La première version trouvée est utilisée.

Recherche/sélection de méthode – résumé

- La variable est accédée.
- L'objet stocké dans la variable est trouvé.
- La classe de l'objet est déterminée.
- La méthode est recherchée dans cette classe.
- Si elle n'y est pas, elle est recherchée dans la superclasse.
- Cela se répète jusqu'à la trouver ou avoir parcouru toute la hiérarchie.
- Les redéfinitions de méthodes sont prioritaires.

Appel `super` dans les méthodes

- Les méthodes redéfinies (les anciennes versions) sont masquées…
- … mais on a souvent besoin de les appeler quand même.
- Une méthode masquée de la superclasse peut encore être appelée :
  - `super.method(...)`
  - Comparez avec l'emploi de `super` dans les constructeurs.

Appeler une méthode redéfinie

```
public class CD
{
    ...
    public void print()
    {
        super.print();
        System.out.println("    " + artist);
        System.out.println("    pistes:" + numberOfTracks);
    }
    ...
}
```

Différences entre `super()` et `super.methode()`

Contrairement à l'appel du constructeur **`super()`**, l'appel de la méthode redéfinie :

- nécessite son nom :

  `super.`*`nom_methode`*`(...)`

- n'est pas nécessairement au début
- n'est pas obligatoire (et n'est pas ajouté automatiquement s'il est absent)
- peut être utilisé dans une autre méthode

Polymorphisme de méthode

- Nous avons analysé la *sélection de méthode polymorphe*.
- Une variable polymorphe peut stocker des objets de différents types pendant l'exécution.
- Les appels de méthode sont polymorphes.
  - La méthode effectivement appelée dépend du type (dynamique) de la variable.

liaison dynamique
(*dynamic binding*)

Problème

- À partir de la liste d'`Items`, comment stocker les `CD` dans une liste séparée ?
- Autrement dit, comment connaître le « vrai » type d'un `Item` (*son type dynamique*) ? Est-il définitivement perdu ?

Solution : L'opérateur `instanceof`

- Utilisé pour déterminer le type dynamique.
- Récupère l'information de type « perdue ».
- Souvent suivi d'un transtypage vers le type dynamique qui a été identifié.

```
// stocker les CD dans une liste séparée
ArrayList<CD> cds = new ArrayList<CD>();
for(Item item : items) {
    if(item instanceof CD) {
        cds.add((CD) item); // cast nécessaire
    }
}
```

Les méthodes de la classe `Object`

- Les méthodes de **`Object`** sont héritées par toutes les classes.
- Chacune peut être redéfinie.
- La méthode **`toString`** est en général redéfinie :
  - **`public String toString()`**
  - Renvoie une représentation de l'objet dans une chaîne de caractères.

Redéfinir `toString` (1)

```
public class Item
{
    ...

    public String toString()
    {
        String line1 = title +
                       " (" + playingTime + " mn)");
        if(gotIt) {
            return line1 + "*\n" + "    " +
                   comment + "\n");
        } else {
            return line1 + "\n" + "    " +
                   comment + "\n");
        }
    }
    ...
}
```

Redéfinir `toString` (2)

- On peut souvent se passer d'une méthode **`print`** explicite dans une classe :
  - `System.out.println(item.toString());`
- Les appels à **`println`** avec un simple objet génèrent des appels à **`toString`** de la classe de l'objet :
  - `System.out.println(item);`

Redéfinir `toString` (2)

- On peut souvent se passer d'une méthode **`print`** explicite dans une classe :
  - `System.out.println(item.toString());`
- Les appels à **`println`** avec un simple objet génèrent des appels à **`toString`** de la classe de l'objet :
  - `System.out.println(item);`

KIF-KIF !

Égalité ≠ Identité
La méthode `equals`

- Quand on dit de deux objets qu'ils sont « égaux », qu'est-ce que cela signifie ?
  – égalité des références (*superficielle*) : objets *identiques*
  – égalité des contenus (*profonde*) : objets *égaux*

- *Rappel :* Pour comparer deux chaînes de caractères (deux objets **`String`**),
  – ne pas utiliser l'opérateur **`==`** qui teste les références
  – utiliser **`equals()`** qui teste les contenus

Redéfinir la méthode **equals**

```
public boolean equals(Object obj)
{
    if(this == obj) {
        return true;
    }
    if(!(obj instanceof CeType)) {
        return false;
    }
    CeType other = (CeType) obj;
    …
    … comparer les champs de this et other
    …
}
```

Redéfinir `equals` dans `Student`

```
public boolean equals(Object obj)
{
    if(this == obj) {
        return true;
    }
    if(!(obj instanceof Student)) {
        return false;
    }
    Student other = (Student) obj;
    return name.equals(other.name) &&
           id.equals(other.id) &&
           credits == other.credits;
}
```

Redéfinir `hashCode`

- Quand on redéfinit equals, on **doit** redéfinir la méthode `hashCode` également.
- Quand deux objets sont égaux, leurs `hashCode` **doivent** donner la même valeur.
- La méthode `hashCode` permet à certaines collections (`HashMap`, `HashSet`...) de gérer les objets efficacement.
- Les EDI professionnels (Eclipse, NetBeans...) ont des outils qui génèrent cette méthode automatiquement.

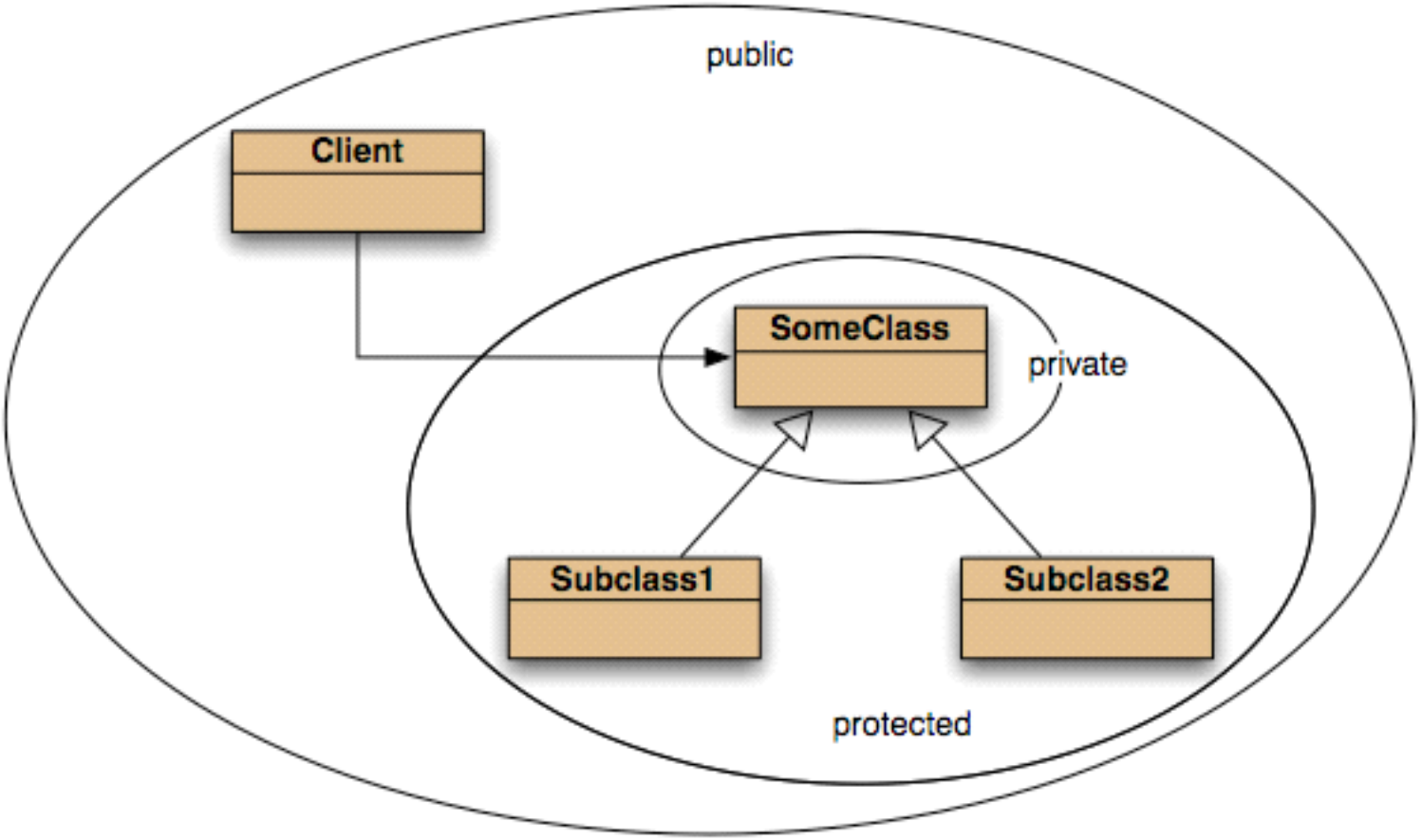
Méthode `hashCode` dans `Student`

```
/**
 * Hashcode technique taken from
 * Effective Java by Joshua Bloch.
 */
public int hashCode()
{
    int result = 17;
    result = 37 * result + name.hashCode();
    result = 37 * result + id.hashCode();
    result = 37 * result + credits;
    return result;
}
```

Accès protégé (`protected`)

- L'accès privé (**`private`**) dans la superclasse peut être trop restrictif pour une sous-classe.
- La relation plus étroite d'héritage est prise en compte par l'*accès protégé* (**`protected`**).
- L'accès protégé est plus restrictif que l'accès **`public`**.
- Nous recommandons cependant de laisser les champs privés (**`private`**).
  - Et de définir des accesseurs et mutateurs protégés.

Niveaux d'accès



Niveaux d'accès

Modificateur d'accès	*Même classe*	*Même paquetage*	*Sous-classe*	*Autres paquetages*
`public`	✔	✔	✔	✔
`protected`	✔	✔	✔	✘
aucun modificateur (accès par défaut)	✔	✔	✘	✘
`private`	✔	✘	✘	✘

Résumé

- Le type déclaré d'une variable est son type statique.
  - Les compilateurs vérifient les types statiques.
- Le type de l'objet qu'elle contient est son type dynamique.
  - Les types dynamiques sont utilisés à l'exécution.
- Les méthodes peuvent être redéfinies dans une sous-classe.
- La recherche de méthode commence avec le type dynamique.
- L'accès *protégé* prend en compte l'héritage.

Sommaire général