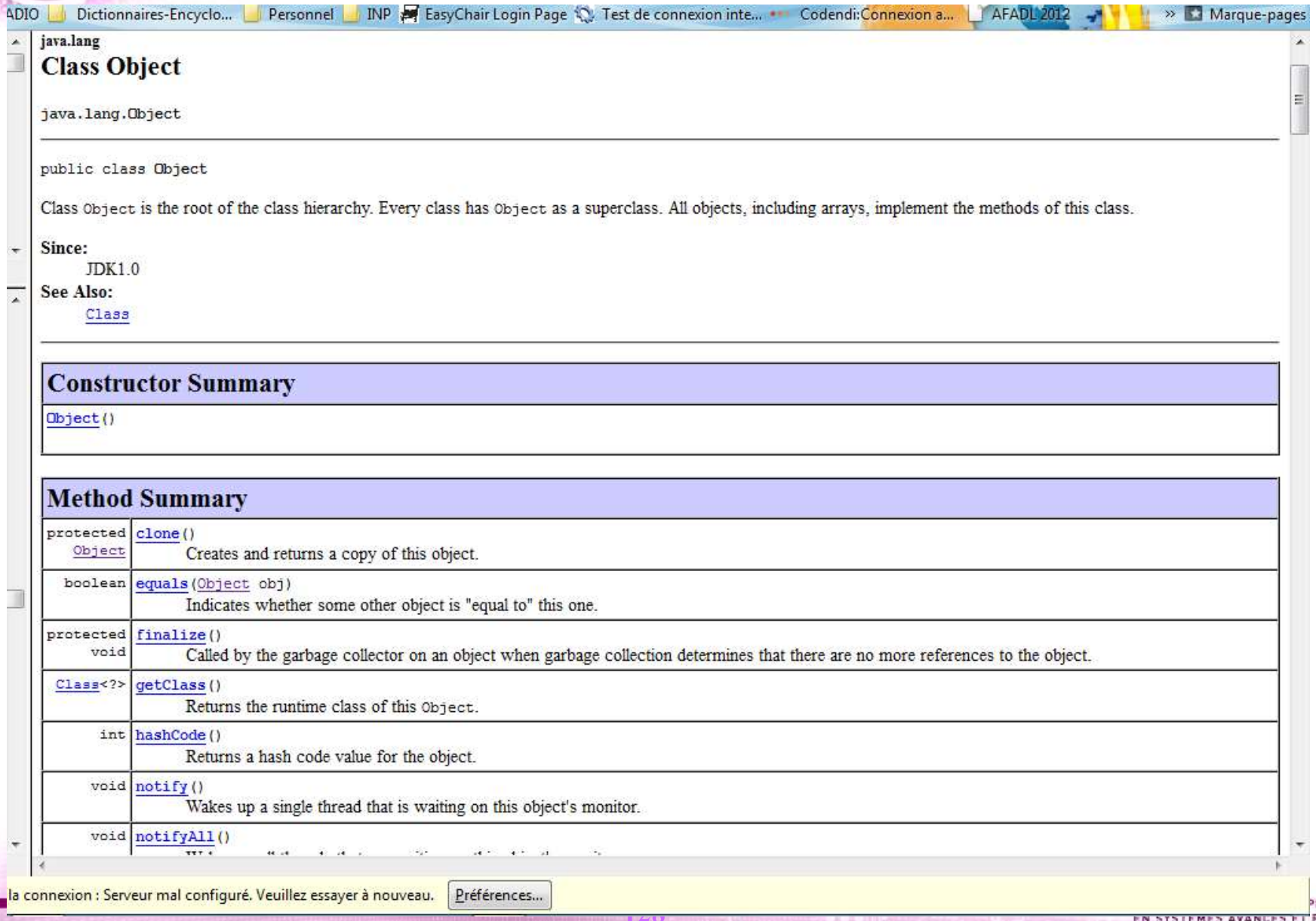
# III. Conception de classes en Java

## III.5 Héritage: questions avancées. Classes abstraites, interfaces.

# La classe Object



ADIO | Dictionnaires-Encyclo... | Personnel | INP | EasyChair Login Page | Test de connexion inte... | Codendi:Connexion a... | AFADL 2012 | Marque-pages

java.lang

## Class Object

java.lang.Object

---

public class Object

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

**Since:**

JDK1.0

**See Also:**

Class

---

### Constructor Summary

| |
|---|
| Object() |

### Method Summary

| | |
|---|---|
| protected Object | clone()<br>Creates and returns a copy of this object. |
| boolean | equals(Object obj)<br>Indicates whether some other object is "equal to" this one. |
| protected void | finalize()<br>Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. |
| Class<?> | getClass()<br>Returns the runtime class of this Object. |
| int | hashCode()<br>Returns a hash code value for the object. |
| void | notify()<br>Wakes up a single thread that is waiting on this object's monitor. |
| void | notifyAll() |

la connexion : Serveur mal configuré. Veuillez essayer à nouveau. Préférences...

## Constructor Summary

| |
|---|
| Object() |

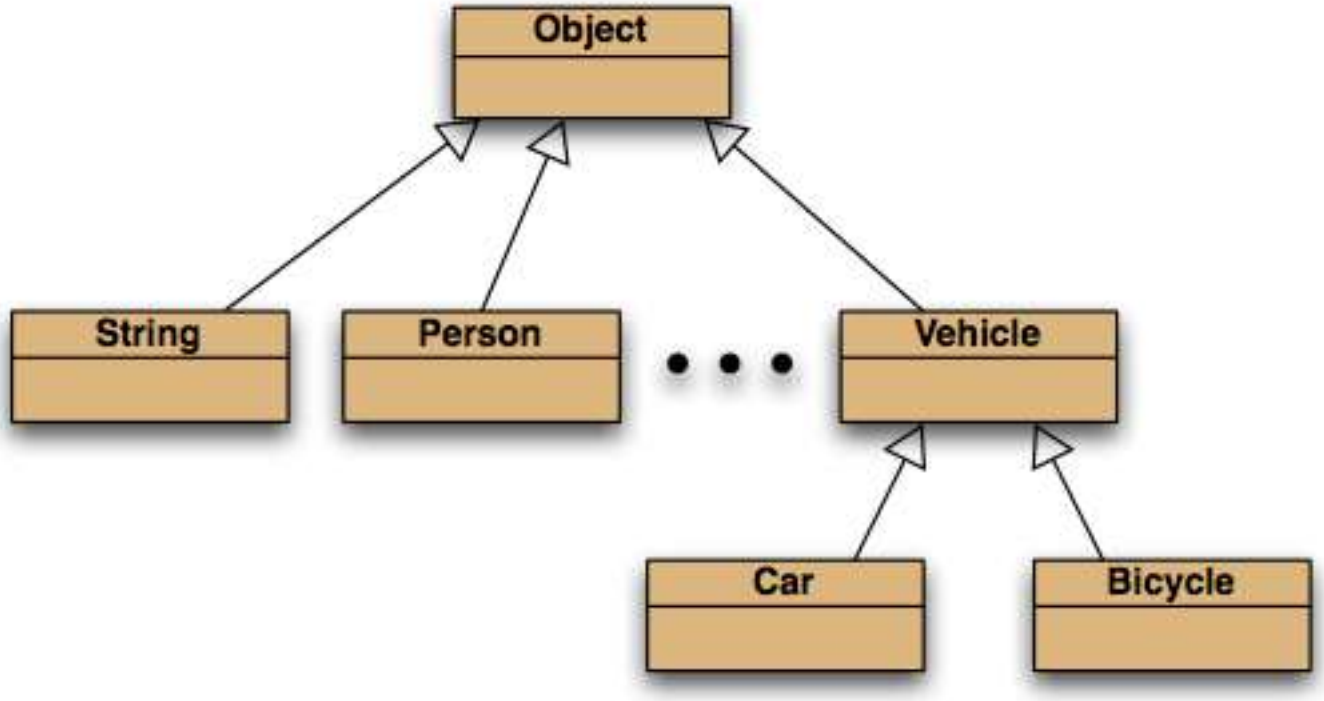
## Method Summary

| | |
|---|---|
| protected Object | clone()<br>Creates and returns a copy of this object. |
| boolean | equals(Object obj)<br>Indicates whether some other object is "equal to" this one. |
| protected void | finalize()<br>Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. |
| Class<?> | getClass()<br>Returns the runtime class of this Object. |
| int | hashCode()<br>Returns a hash code value for the object. |
| void | notify()<br>Wakes up a single thread that is waiting on this object's monitor. |
| void | notifyAll()<br>Wakes up all threads that are waiting on this object's monitor. |
| String | toString()<br>Returns a string representation of the object. |
| void | wait()<br>Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object. |
| void | wait(long timeout)<br>Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed. |
| void | wait(long timeout, int nanos)<br>Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed. |

rs de la connexion : Serveur mal configuré. Veuillez essayer à nouveau. Préférences...

# La classe Object

*Toutes les classes héritent d'***Object**.



# Méthodes de la classe Object

- Les méthodes de `Object` sont héritées par toute classe
- Une sous-classe peut redéfinir les méthodes dont elle hérite.
- Exemple fréquent de redéfinition :
  - `public String toString(){…}`

# Exemple de redéfinition de méthode

```
public class Salle {
 private int capacité;
 private String nom;

 public Salle(int c, String n){
  capacité = c;
  nom = new String(n);
 }

 public String toString(){
  return nom + " ("
    + capacité + " places)";
 }

 public int getCapacité(){
  return capacité;
 }

 public String getNom(){
  return nom;
 }
}
```

```
public class SalleCTD extends Salle{
 public SalleCTD(int capacité, String nom){
  super(capacité, nom);
 }

  public String toString(){
   return "Salle cours-TD "+
           super.toString();
  }
}
```

```
...
SalleCTD s = new SalleCTD(180, "D030");
System.out.println(s);
```

*« Salle de cours-TD D030 (180 places) »*

# Types statiques, types dynamiques

```
public class Salle {
 private int capacité;
 private String nom;
 ...

 public String toString(){
  return nom + " ("
    + capacité + " places)";
 }
 ...
}
```

```
public class SalleCTD extends Salle{
 public SalleCTD(int capacité, String nom){
  super(capacité, nom);
 }

  public String toString(){
   return "Salle cours-TD "+
           super.toString();
  }
}
```

```
...
Salle s1 = new SalleCTD(180, "D30");
SalleCTD s2 = new SalleCTD(100, "A042");
...
s1 = new Salle(80, "C04");

System.out.println(s1 + s2 +s3);
// Equivaut à afficher (s1.toString() + s2.toString() +s3.toString())
...
```

***Quelles méthodes toString sont appelées?***

# Type statique – type dynamique

Type de s2?

```
SalleCTD s2 = new SalleCTD(100, "A042");
```

Type de s1?

```
Salle s1 = new SalleCTD(180, "D30");
```

# Type statique vs. type dynamique

- Type statique = type déclaré de la variable
- Type dynamique = type d'un objet à l'exécution
- Le compilateur vérifie la compatibilité des types statiques
- La compatibilité des types dynamiques est vérifié à l'exécution

# Type statique – type dynamique

Type de s2?
SalleCTD

```
SalleCTD s2 = new SalleCTD(100, "A042");
```

Type de s1?
Statique : Salle
Dynamique : SalleCTD

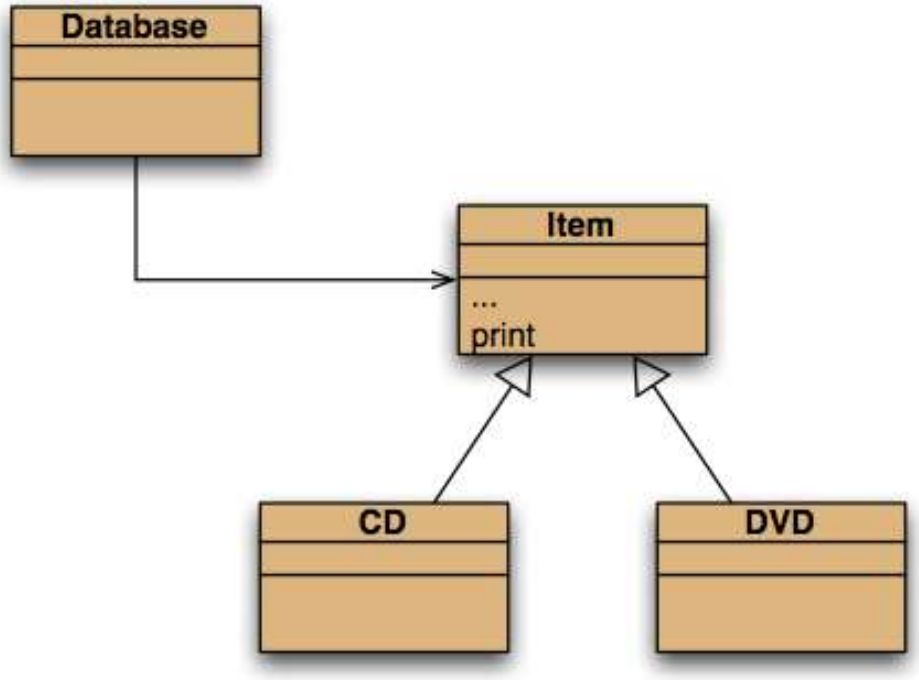
```
Salle s1 = new SalleCTD(180, "D30");
```

# Types statiques, types dynamiques

Retour à l'application DoME.
`Database` contient une liste d'`Item`

`print` : affiche les attributs de l'objet de la classe `Item`

Ici, `print` n'est défini que dans la classe `Item`



# Affichage des Item de Database

```
public void list()
{
    for(Item item : items) {
        item.print();
    }
}
```

Ce qu'on veut

```
CD: A Swingin' Affair (64 mins)*
    Frank Sinatra
    tracks: 16
    my favourite Sinatra album

DVD: O Brother, Where Art Thou? (106 mins)
     Joel & Ethan Coen
     The Coen brothers' best movie!
```

Ce qu'on a

```
title: A Swingin' Affair (64 mins)*
       my favourite Sinatra album

title: O Brother, Where Art Thou? (106 mins)
       The Coen brothers' best movie!
```

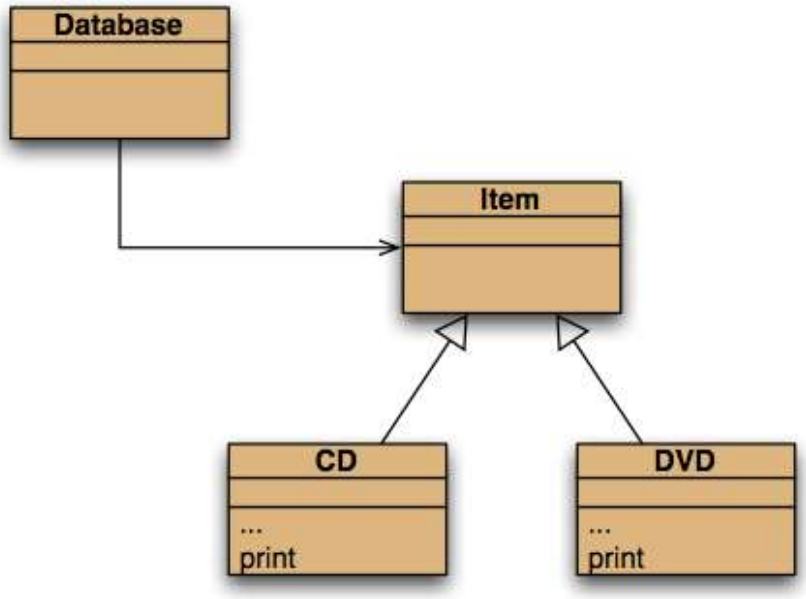
# Pourquoi?

- La méthode `print` de `Item` n'imprime que les attributs d'`Item`.

- L'héritage est à sens unique
  - Une sous-classe hérite (donc connaît) des attributs et méthodes de sa super-classe.

  - La super-classe ne sait rien de ses sous-classes.

# Que faire?

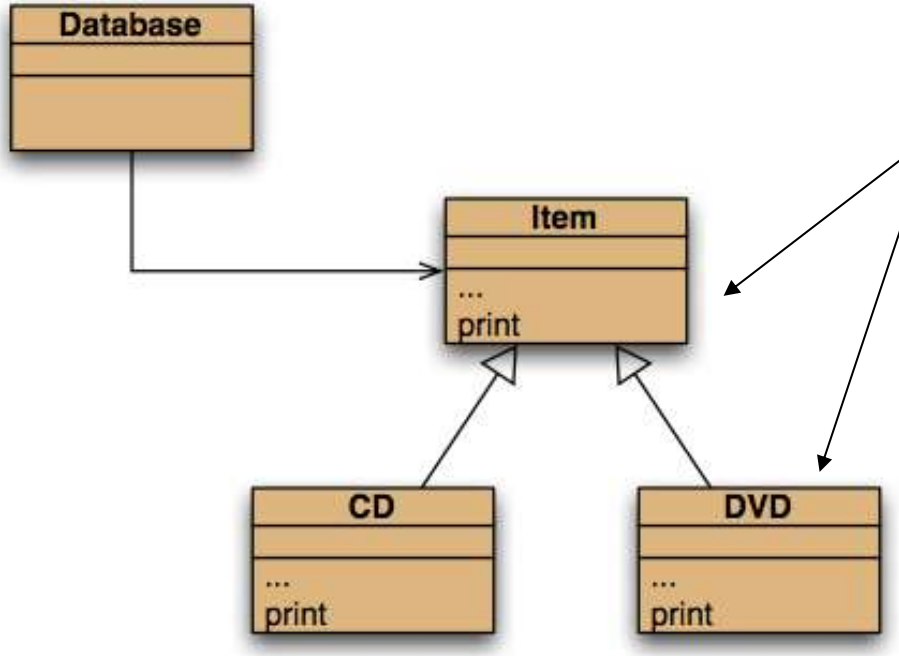



```
public void list()
{
    for(Item item : items) {
        item.print();
    }
}
```

**Erreur de compilation : print n'est pas défini**

- Définir **`print`** au niveau des sous-classes?
  - Chaque sous-classe a sa propre définition de la méthode.
  - Mais les attributs d'**`Item`** sont privés… (`print` ne peut pas y accéder)
  - et `Database` ne manipule que des `Item` … donc ne peut pas trouver de méthode `print` dans `Item`.

# La solution




`print` défini dans la super-classe ET dans les sous-classes.

Type statique et type dynamique : ok

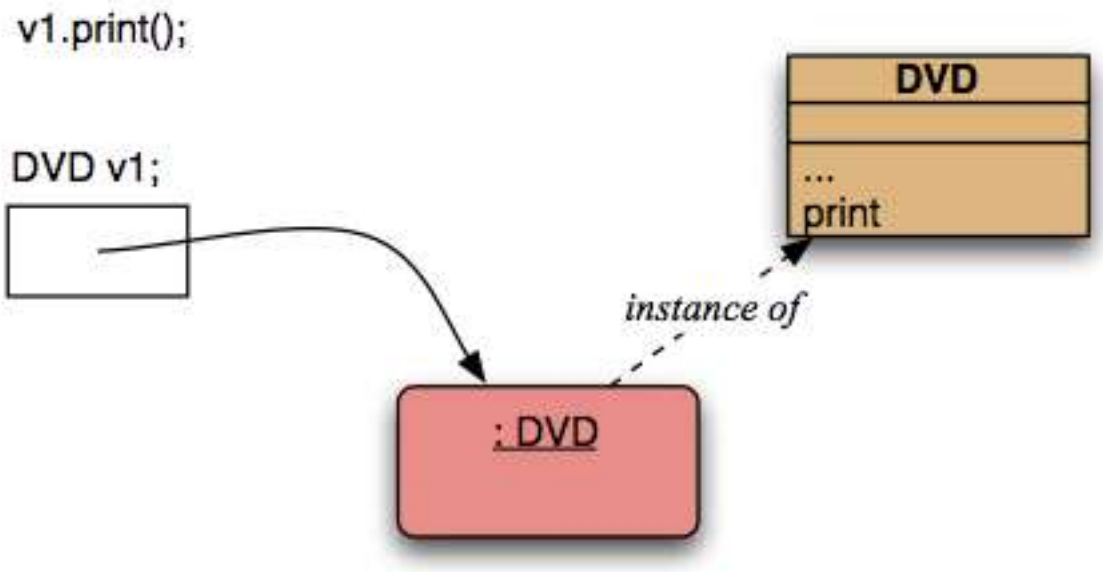
```
public void list()
{
    for(Item item : items) {
        item.print();
    }
}
```

**Compilation ok : print défini dans Item (type statique)**
**Exécution ok : c'est la méthode print de CD ou DVD qui est appelée (type dynamique)**

# Redéfinition de méthodes

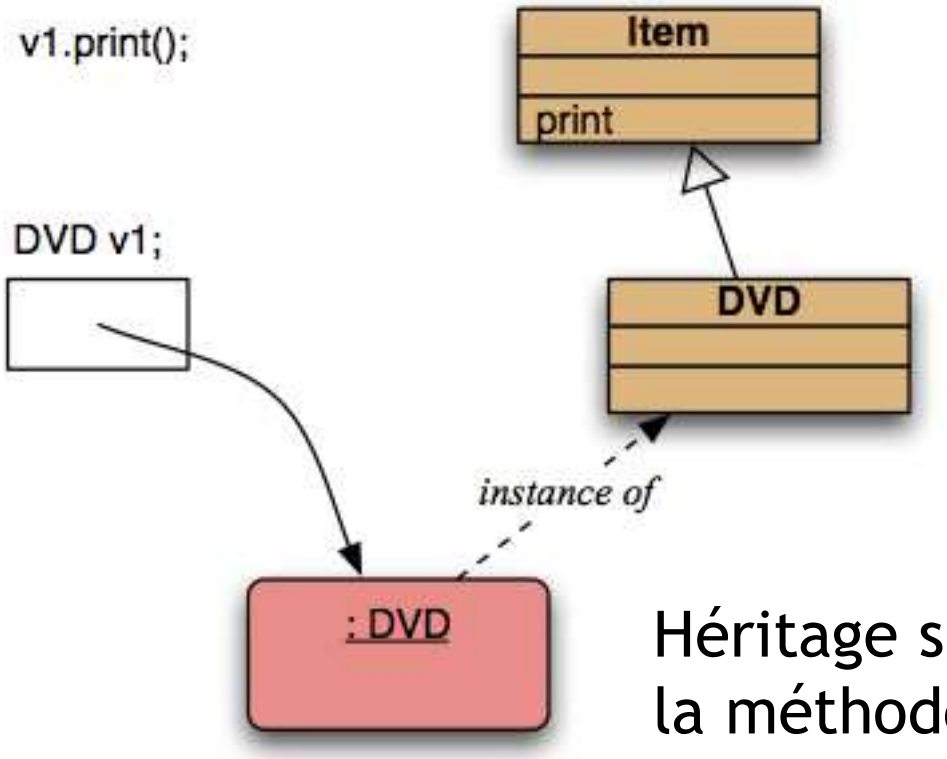
- La super-classe et la sous-classe définissent une méthode avec la même signature
- A l'exécution, c'est la méthode de la sous-classe qui est appelée.
- Que devient la méthode de la super-classe?

# Recherche de méthode



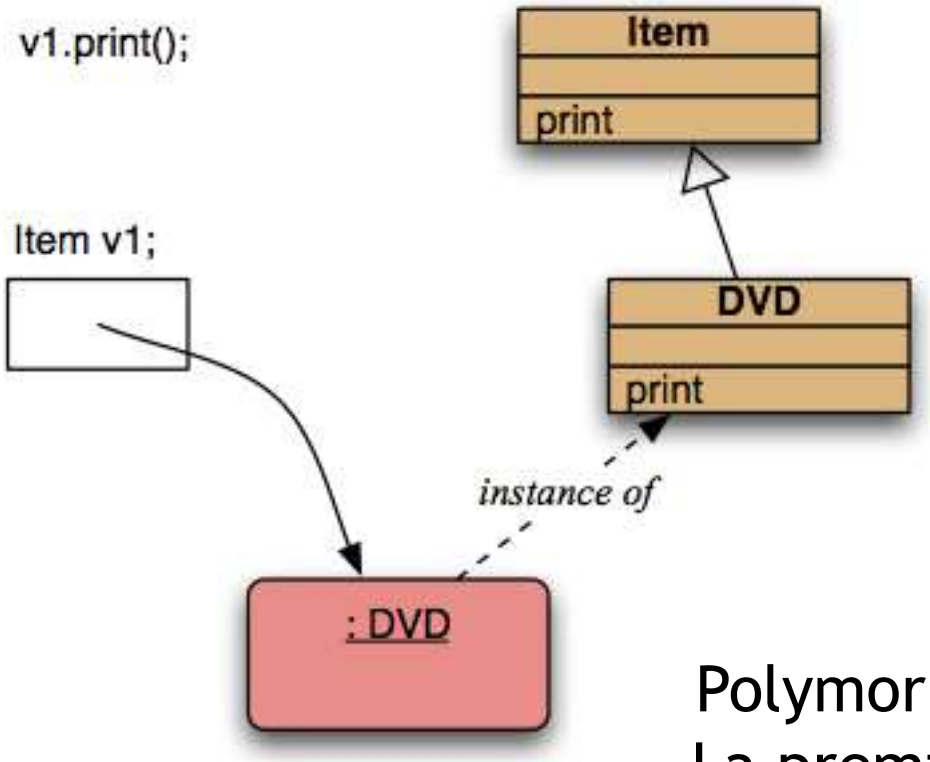

Pas d'héritage ou polymorphisme
Choix évident

# Recherche de méthode




Héritage sans overriding:
la méthode est recherchée dans la hiérarchie (la méthode print de Item est choisie)

# Recherche de méthode




Polymorphisme+overriding:
La première (en partant du bas) méthode est choisie.

# Polymorphisme de méthodes

- Les appels de méthodes sont polymorphes
  - La méthode appelée sur un objet est déterminée en fonction du type de l'objet à l'exécution

# Retour sur la classe Document et la Livrothèque (TD2)

**/* Comment afficher la liste des auteurs des ouvrages ayant un auteur */**

```
public class Bibliothèque{
        private List<Document> ld;

        public Bibliothèque(){
                ld = new ArrayList<Document>();
        }

        // ...
```

**/* Version avec instanceof : à proscrire - la classe Document n'a pas à connaître ses sous-classes */**

```
public void afficherAuteurs(){
                for(Document d : ld){
                        if (d instanceof Livre){
                                System.out.println(d);
                        }
                }
        }
```

# Retour sur la classe Document et la Livrothèque (TD2)

```
public class Bibliothèque{
   private List<Document> ld;

   public Bibliothèque(){
      ld = new ArrayList<Document>();
   }
...
   public void afficherAuteurs(){
      for(Document d : ld){
         if (d.hasAuteur()){
            System.out.println(d.getAuteur());
         }
      }
   }

}

public class Document{
   public boolean hasAuteur(){
      return false;
   }

   public String getAuteur(){
      return ("");
   }

}
```

```
public class Livre extends Document{
   private String auteur;
          ...
   public boolean hasAuteur(){
      return true;
   }

   public String getAuteur(){
      return auteur;
   }
}
```

# Retour sur la classe Document et la Livrothèque (TD2)

```
public class Livrotheque extends Bibliotheque {

    public Livrotheque(int capacite) {
        super(capacite);
    }

    boolean ajouter(Livre doc) {
        System.out.println("ajouter Livre");
        return super.ajouter(doc);
    }

    boolean ajouter(Document doc) {
        System.out.println("ce n'est pas un livre");
        return false;
    }
}

public class Main {

        public static void main (String[] args){

        int cap = 10;
        Livrotheque l = new Livrotheque(cap);

        l.ajouter(new Livre(1, "Mon livre à moi", "Moi-même", 150));
        l.ajouter(new Revue(1, "Ma revue", 2019, 3));
        l.ajouter(new Manuel(1, "Math", "Newton", 150, 3));
    }
}
```

# "protected"

- Reste la question : comment la méthode `print` des classes `CD` ou `DVD` peut accéder aux attributs de la classe `Item`?
  - A priori, elle ne peut pas et il n'y a aucune raison pour qu'elle puisse!

- Attribut ou méthode `protected`
  - Peut être utilisé (accès/modification) par la classe et ses sous-classes uniquement.

  - `private` : uniquement dans la classe
  - `protected` : dans la classe et ses sous-classes
  - `public` : dans toute classe

- Conseils d'utilisation
  - Attributs `public` : à proscrire!

  - Attributs `protected` : à éviter
    - Définir des mutateurs/accesseurs `protected`

# Méthodes statiques

Méthode statique

- Méthode qui peut être appelée sans créer un objet de sa classe ("méthode de classe")
- Une méthode statique ne peut accéder qu'à des attributs ou à des méthodes statiques

- Exemples
  - Méthode `main`
  - `Java.lang.Math.sqrt(x)`

# Classes et méthodes abstraites

- Mot clé `abstract` dans la signature d'une méthode
  - La méthode est "abstraite"
  - Elle n'a pas d'implémentation (corps)
- Mot clé `abstract` dans la definition d'une classe
  - La classe est "abstraite"
  - On ne peut pas créer d'objets à partir d'une classe abstraite
  - Toute classe disposant de méthodes abstraites est abstraite
  - Les classes abstraites sont destinées à être héritées
    - Les sous-classes définissent l'implémentation des méthodes abstraites

# Classes abstraites : à quoi ça sert? (1)

## *Créer une obligation pour les sous-classes*

```
abstract public class Salle {
 private int capacité;

 public Salle(int c){
  capacité = c;
}

 public String toString(){
  return " ("
    + capacité + " places)";
 }

 public int getCapacité(){
  return capacité;
 }

 abstract public String getNom();
}
```

```
public class SalleCTD extends Salle{
 private String nom;

 public SalleCTD(int capacité,
                 String nom){
  super(capacité);
  this.nom = nom;
 }

  public String toString(){
   return "Salle cours-TD "+ nom + " " +
           super.toString();
 }

 public String getNom(){
  return nom;
 }

}
```

# Classes abstraites : à quoi ça sert? (2)

## *Simplement interdire la creation d'objets*

```
abstract public class Salle {
 private int capacité;
 private String nom;

 public Salle(int c, String n){
  capacité = c;
  nom = new String(n);
 }

 public String toString(){
  return nom + " ("
    + capacité + " places)";
 }

 public int getCapacité(){
  return capacité;
 }

 public String getNom(){
  return nom;
 }
}
```

```
public class SalleCTD extends Salle{
 public SalleCTD(int capacité, String nom){
  super(capacité, nom);
 }

  public String toString(){
   return "Salle cours-TD "+
           super.toString();
 }
}

public class SalleTP extends Salle{
 private Discipline type;
 public SalleTP(int capacité, String nom,
                Discipline d){
  super(capacité, nom);
  type = d;
 }

 public String toString(){
  return "Salle TP "+ type + " " +
                super.toString();
 }
}
```

# Interfaces

- Une interface n'est pas une classe mais une *spécification* qu'une classe doit satisfaire.

```
public interface Véhicule{
        public void faireLePlein();
        public int getVitesseMax();
}

public class Vélo implements Véhicule{
        public Vélo(){}
        public void faireLePlein(){
                System.out.println("un vélo est toujours chargé!");
        }
        public int getVitesseMax(){
                return 30;
        }
}

public class Auto implements Véhicule{
        private int v;

        public Auto(int vitesseMax){
                this.v = vitesseMax;
        }
        public void faireLePlein(){
                System.out.println("Je vais à la pompe");
        }
        public int getVitesseMax(){
                return this.v;
        }
}
```

# Interfaces

- Les classes implémentant une interface n'héritent pas de son code (l'interface n'en a pas) …
  - … mais elles sont des sous-types de l'interface (le polymorphisme marche)

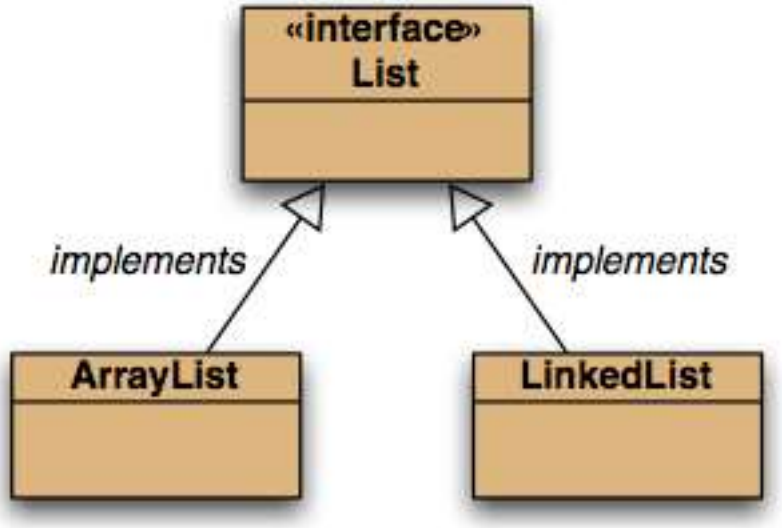
```
Véhicule v1 = new Vélo();
Véhicule v2 = new Auto(150);
v2.fairelePlein();
```

  - On ne peut pas créer d'"objet interface"

```
Véhicule v3 = new Véhicule(); // erreur
```

- Une interface n'a pas de constructeur.
- Toutes les méthodes d'une interface sont abstraites.
- Toutes les méthodes d'une interface sont publiques.
- Tous les attributs sont `public`, `static`, `final`.

# Exemple d'implémentations multiples




```
List<String> l1 = new ArrayList<String>();
List<String> l2 = new Linkedlist<String>();
...
l1.add(l2.get(i));
```

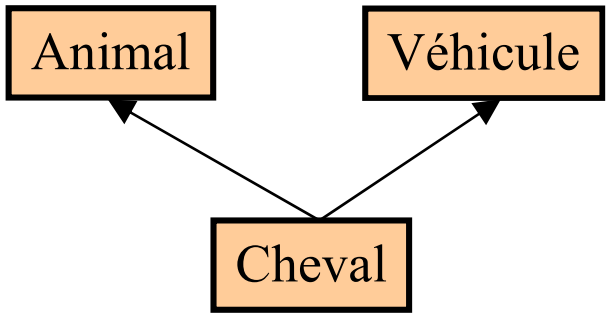
# Intérêt des interfaces

- Une utilisation typique des interfaces :

```
public class Parking{
    …
    public void garer(Véhicule v){
    /* Cette méthode traitera tout objet issu d'une classe
       implémentant l'interface Véhicule */
    …
    }
}
```

- Autre utilisation typique
  - *"Héritage multiple"*

# Héritage multiple




- Dilemme : si les classes `Animal` et `Véhicule` possèdent toutes les deux une méthode `alimenter`, laquelle doit être héritée par la classe `Cheval`?

- En java l'héritage multiple est interdit (au sens du « extends »)…
  - … mais on peut (presque) le faire avec des interfaces

```
class Cheval extends Animal implements Véhicule
class Cheval extends Animal implements Véhicule, Mammifère, Quadrupède
```

# Bilan des concepts introduits

- Classe Object
- Redéfinition de méthodes
- Type statique et type dynamique d'un objet
- `protected`
- Méthodes `static`
- Classes abstraites, méthodes abstraites
- Interfaces
- Interfaces et héritage multiple