# Héritage

Conception Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1

# Réutiliser...

c'est un des (le ?) soucis constants des programmeurs

- le programmeur d'API ("pour programmeur") :
  - permettre la réutilisation au maximum
    ↪ faire une "vraie" API, réellement (ré-) utilisable
  - faciliter le travail de réutilisation
- le programmeur pour client :
  diminuer le volume de travail à réaliser
  - ne pas refaire ce qui a été fait
  - diminuer les sources d'erreurs (les API ont a priori été testées et validées)

**un bon programmeur est un programmeur paresseux** ($\neq$ fainéant !)

programmer bien tout de suite, pour avoir à en faire moins plus tard

**Programmer c'est investir !**

# ... un type

*Définir un comportement propre à son contexte d'utilisation tout en respectant des interfaces prédéfinies et s'insérer ainsi dans un cadre préétabli.*

notion de "framework"

en JAVA : utilisation des `interfaces`

cf. POO

# ... un comportement

(par agrégation/composition)

réutilisation partielle et nécessitant une contextualisation

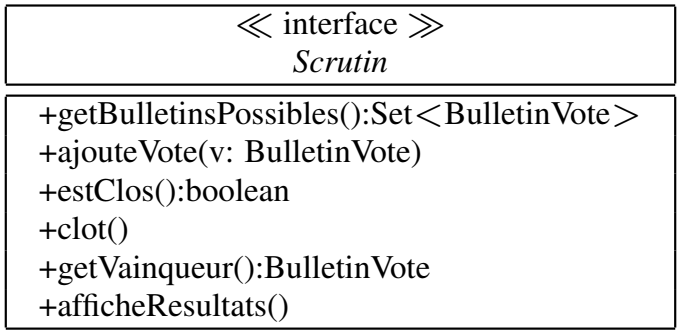
**adaptation**

- définir un attribut de la classe dont on veut récupérer le comportement
- définir des méthodes correspondant aux comportements que l'on veut récupérer et en les ajustant à son contexte
- le corps de ces méthodes consiste en un appel des méthodes correspondantes de l'attribut avec une adaptation éventuelle

```
public interface Strategie {
   public Coup choisitCoup();
   public int valeurCoup();
}
public class Joueur {
   private Strategie strategie;
   public Joueur(Strategie s) {
      this.strategie = s;
   }
   public Coup joue() {
      this.strategie.choisitCoup();
   }
   public float valeurCoup() { // adaptation
      return (float) this.strategie.valeurCoup();
   }
}
```

- **Réutiliser un type :** on s'assure de la conformité de type de la classe créée et on peut utiliser le polymorphisme sur les instances,
  **mais** on doit réécrire pour chaque classe implémentant l'interface le code de toutes les méthodes y compris si celui-ci est le même pour plusieurs classes
  ↪ génant dans le cas d'une modification qu'il faut alors reporter plusieurs fois
- **Réutiliser un comportement :** on n'a pas besoin de réécrire le code des méthodes
  **mais** on a pas de compatibilité de type et donc de polymorphisme, les instances de la classe créée ne sont pas du type de la classe de l'attribut

# Factorisation du code ?




```
public class ScrutinMajoritaire
                implements Scrutin {
   private boolean estClos;
   public boolean estClos() {
      return this.estClos;
   }
   ...
   public BulletinVote getVainqueur() {
      ...  TRAITEMENT ...
   }
}
```

```
public class ScrutinRelatif
                implements Scrutin { // même type
   private boolean estClos;
   public boolean estClos() {
      return this.estClos;
   }
   ...// et d'autres répétitions
   public BulletinVote getVainqueur() {
      ...  TRAITEMENT DIFFERENT ...
   }
}
```

- attributs et codes *répétés* ! $\Longrightarrow$ **factorisation de code** possible
- mais `getVainqueur` *différents*...

comment concilier la factorisation de code et les différences ?

# Héritage

## Héritage de classe

On peut définir une classe **héritant** d'une autre classe.

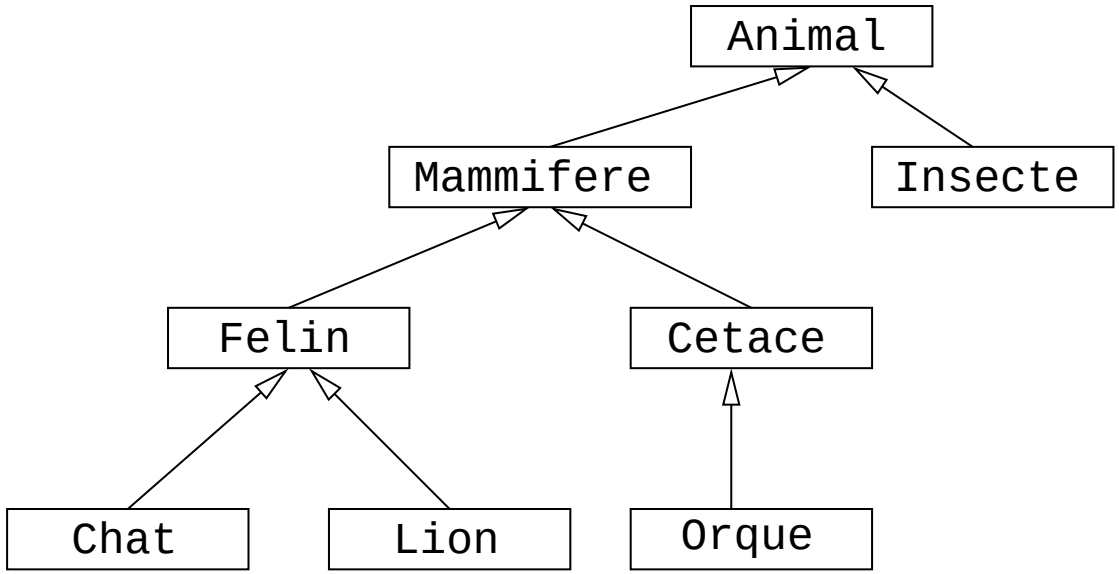
- la classe héritante
  - **récupère tous** les comportements (accessibles) de la classe dont elle hérite,
  - **peut modifier** certains comportements hérités,
  - **peut ajouter** de nouveaux comportements qui lui sont propres.
- les instances de la classe héritante sont également du type de la classe héritée : **polymorphisme**
  ↪ on parle de **sous-classe** (donc sous-type)
  ↪ une instance de sous-classe est également du type de la classe mère (ou **super-classe**).

## relation *est-un*

on fait souvent référence à l'héritage comme réalisant la relation "*est un*"

Par exemple :

- un `Mammifere` *est un* `Animal` vertébré, à température constante, respirant par des poumons
- un `Insecte` *est un* `Animal` terrestre invertébré, à 6 pattes, le plus souvent ailé, respirant par des trachées
- un `Felin` *est un* `Mammifere` terrestre carnivore
- un `Cetace` *est un* `Mammifere` qui vit dans l'eau
- un `Chat` *est un* `Felin` qui miaule
- un `Lion` *est un* `Felin` à pelage fauve qui rugit
- un `Orque` *est un* `Cetace` carnivore




- *Par héritage*, une instance de `Chat` est aussi un `Felin`, un `Mammifere` et un `Animal`
- l'interface publique définie dans `Felin` fait partie de l'interface publique d'un objet `Chat`.
- idem avec les interfaces publiques de `Animal` et `Mammifere`.

*rappel : interface publique = ensemble des messages publics*

# extends

## extends

En JAVA pour indiquer qu'une classe hérite d'une autre, on utilise le mot-clé **extends**.

```
public class Animal {
    ...
}
public class Mammifere extends Animal {
    ...
}
public class Felin extends Mammifere {
    ...
}
public class Chat extends Felin {
    ...
}
```

```
public class Insecte extends Animal {
  ...
}
public class Cetace extends Mammifere {
  ...
}
public class Lion extends Felin {
  ...
}
public class Orque extends Cetace {
  ...
}
```

## Héritage simple

En JAVA, on ne peut hériter que d'une classe à la fois.

# Object : des mystères révélés

toutes les classes héritent par défaut de la classe `Object`

(soit directement soit via leur superclasse)

donc

- tout objet peut se faire passer pour un objet de type `Object`
  ↪ collections
- tout objet peut utiliser les méthodes définies par la classe `Object`

*exemples* : `equals(Object o)`, `toString()`, `hashCode()`

## Factorisation du comportement

Les comportements (**accessibles**) définis dans une classe sont **automatiquement disponibles** pour les instances des classes qui en héritent (même indirectement).

```
public class Mammifere extends Animal {
   public String organeDeRespiration() {
      return "poumons";
   }
}
public class Felin extends Mammifere { ... }

Felin felix = new Felin();                    // un Felin peut utiliser les
String s = felix.organeDeRespiration();// méthodes publiques de ses
                                              // super-classes

Mammifere mamm = felix;              // upcast autorisé vers Mammifere
Animal an = felix;                   // ou vers Animal
```

- factorisation également au niveau de l'"état"
  - les attributs des super-classes sont des attributs de la classe héritante

```
public class Animal {                    // sous-entendu :  "extends Object"
   public Habitat habitat;
}
public class Mammifere extends Animal {
}
public class Felin extends Mammifere {
}


// utilisation ...
Felin felix = new Felin();
felix.habitat = Habitat.TERRESTRE;// habitat est un attribut de Felin
```

# Mais les attributs privés ne sont toujours pas accessibles

```
public class Animal {
   private String name;
   public String getName(){
      return this.name;
   }
   public void setName(String name){
      this.name = name;
   }
}

public class Mammifere extends Animal {
   public void someMethod() {
      System.out.println(this.name);          // NON ! private :  accès interdit
      this.setName("un nom");                 // invocations légales, l'attribut
      System.out.println(this.getName());     // name existe donc bien pour
   }                                          // Mammifere
}
```

# Extension

- la sous-classe peut **ajouter** des nouveaux comportements
- la classe héritante est donc une **extension** de la classe héritée

```
public class Mammifere extends Animal {
   public String organeDeRespiration() {
      return "poumons";
   }
}
public class Felin extends Mammifere {
   public int getNbDePattes() {
      return 4;
   }
}
```

- un objet `Felin` peut invoquer `organeDeRespiration` et `getNbDePattes`

$$\textbf{Felin} \subset \textbf{Mammifere} \subset \textbf{Animal}$$

au sens où un `Felin` peut se faire passer pour un `Mammifere` ou un `Animal` : tout message envoyé à un `Mammifere` peut l'être à un `Felin`

# Spécialisation

- une classe héritante peut **redéfinir** un comportement défini dans une super-classe
  c'est ce comportement qui est utilisé par ses instances
- on parle de **surcharge** de méthode
  (NB : même signature, sinon extension)

```
public class Mammifere extends Animal {
   public int getNbDePattes() {
      return 4;
   }
}
public class Cetace extends Mammifere {
   public int getNbDePattes() {
      return 0;
   }
}
// utilisation
Cetace cet = new Cetace();
System.out.println(cet.getNbDePattes());   // affiche 0
Mammifere mam1 = new Mammifere();
System.out.println(mam1.getNbDePattes());  // affiche 4
Mammifere mam2 = cet;
System.out.println(mam2.getNbDePattes());  // affiche 0        "late binding"
```

# Attributs : masquage

```
public class Mammifere extends Animal {
   public int nbDePattes = 4;
public class Cetace extends Mammifere {
   public int nbDePattes = 0;
}

Cetace cet = new Cetace();
System.out.println(cet.nbDePattes);          // affiche 0
```

Mais il s'agit ici d'un **masquage** d'attribut, les 2 continuent d'exister.

NB : il est possible de masquer en changeant de type

# Attention !!!

```
public class Mammifere extends Animal {
   public int nbPattes = 4;
}
public class Cetace extends Mammifere {
   public int nbPattes = 0;
}


// utilisation...
Cetace cet = new Cetace();
System.out.println(cet.nbPattes); // ↦ ??0
Mammifere mam = cet;
System.out.println(mam.nbPattes); // ↦ ??4 !!!
```

# Attention !!!

```
public class Mammifere extends Animal {
   private int nbPattes = 4;
   public int getNbPattes() {
      return this.nbPattes;
   }
}

// utilisation...
public class Cetace extends Mammifere {
   private int nbPattes = 0;
}

Cetace cet = new Cetace();
System.out.println(cet.getNbPattes()); // ↦ ??4 !!!
```

## Moralité

**Eviter** les surcharges d'attributs ! (sens ?)

# super

- réutiliser le traitement réalisé par la super-classe pour une méthode surchargée ?
  utiliser la référence **super** pour invoquer la méthode de la super classe

```
public class Mammifere extends Animal {
   public String uneMethode() {
      return "mammifere";
   }
}
public class Cetace extends Mammifere {
   public String uneMethode() { ... }     //surcharge
   public String autreMethode() {
      return super.uneMethode()+" marin";
   }
}
S.o.p(new Cetace().autreMethode()); // affiche :  mammifere marin
```

`super` : une sorte de `this` pour ma "sous-couche" (= super-classe)

# Héritage d'interfaces

- il est possible de définir une interface héritant d'une autre interface.
- comme pour les classes, on peut *étendre* par héritage.

```
public interface Aquatique {
   public void nage();
}
public interface Marin extends Aquatique {
}
public interface Predateur {
   public void chasse();
}
public interface PredateurMarin extends Marin, Predateur {
   public float kiloPoissonsMangesParJour();
}
```

- intérêt : typage mutiple = intersection de types.

## Héritage d'interfaces

```
public interface Animal { }
public interface Aquatique { }
public interface AnimalAquatique extends Animal, Aquatique {}
public class Zoo {
   public void addAnimal(Animal animal) { ...  }
}
public void Aquarium {
   public void add(Aquatique aquatique) { ...  }
}
public class ZooAvecAquarium extends Zoo {
   protected Aquarium aquarium;
   public void addAnimalAquatique(AnimalAquatique animalAqua) {
      this.addAnimal(animalAqua);
      this.aquarium.add(animalAqua);
   }
}
public class Mammifere implements Animal { ...}
public class Cetace extends Mammifere implements AnimalAquatique { ...  }

public class Nenuphar implements Aquatique { ...  }
   // ...  utilisation ...
ZooAvecAquarium zoo = new ZooAvecAquarium();
Cetace cet = new Cetace();
zoo.addAnimalAquatique(cet);
zoo.getAquarium().add(new Nenuphar());
```

# final

- il est possible d'interdire l'héritage d'une classe
- il suffit de mentionner le modificateur `final` dans l'entête de déclaration de classe.

```
public final class ClassePasHeritable {
    ...
}
```

`java.lang.String`, `java.lang.Boolean`, `java.net.InetAddress`, etc.

- il est possible d'interdire la surcharge (redéfinition) d'une méthode
- il suffit de mentionner le modificateur **`final`** dans la signature de la méthode.

```
public class UneClasse {
   public final void uneMethode() { ...  }
}
public class UneSousClasse extends UneClasse {
   // ne peut redéfinir public void uneMethode()
}
```

dans classe `java.applet.Applet`

```
public final void setStub(AppletStub stub)
public static final AudioClip newAudioClip(URL url)
```

**`final`** ∼ "*constant*"

# Application

Garantir le code exécuté indépendamment de tout héritage possible.

Dans une classe `Scrutin`. (sujet COO juin 2006)
Garantir l'anonymat du votant quelque soit le type de `Scrutin` :

```
public final void vote(Votant votant)
            throws VoteImpossibleException {
   if (this.peutVoter(votant)) {
      Vote vote = votant.vote(this.lesVotesPossibles);
      this.enregistreVote(vote);
      this.aVote(votant);
   }
   else {
      throw new VoteImpossibleException ();
   }
}
private final void enregistreVote(Vote vote) {
   this.lesVotes.add(vote);
}
```

# protected

- nouveau modificateur d'accès : **protected**
  offrir l'accès aux instances des sous-classes sans rendre public

```
public class Animal {
   protected String name;
   public String getName(){
      return name;
   }
   public void setName(String name){
      this.name = name;
   }
}
public class Mammifere extends Animal {
   public void accesLegal() {
      this.name = "un nom de mammifere";// accès légal, sous-classe et protected
   }
}

public class Quelconque {
   public void illegal() {
      Animal animal = new Animal();
      animal.name = "un nom";            // accès interdit, ne compile pas
      animal.setName("un nom");          // accès légal
   }
}
```

# Modificateurs d'accès

| *modificateur**accès* | classe | classe héritée | même paquetage | autre cas |
|---|---|---|---|---|
| `private` | × | – | – | – |
| `protected` | × | × | × | – |
| aucun (“*friendly*”) | × | – | × | – |
| `public` | × | × | × | × |

`private` *for my eyes only*

`protected` pour mes descendants et mes amis (!)

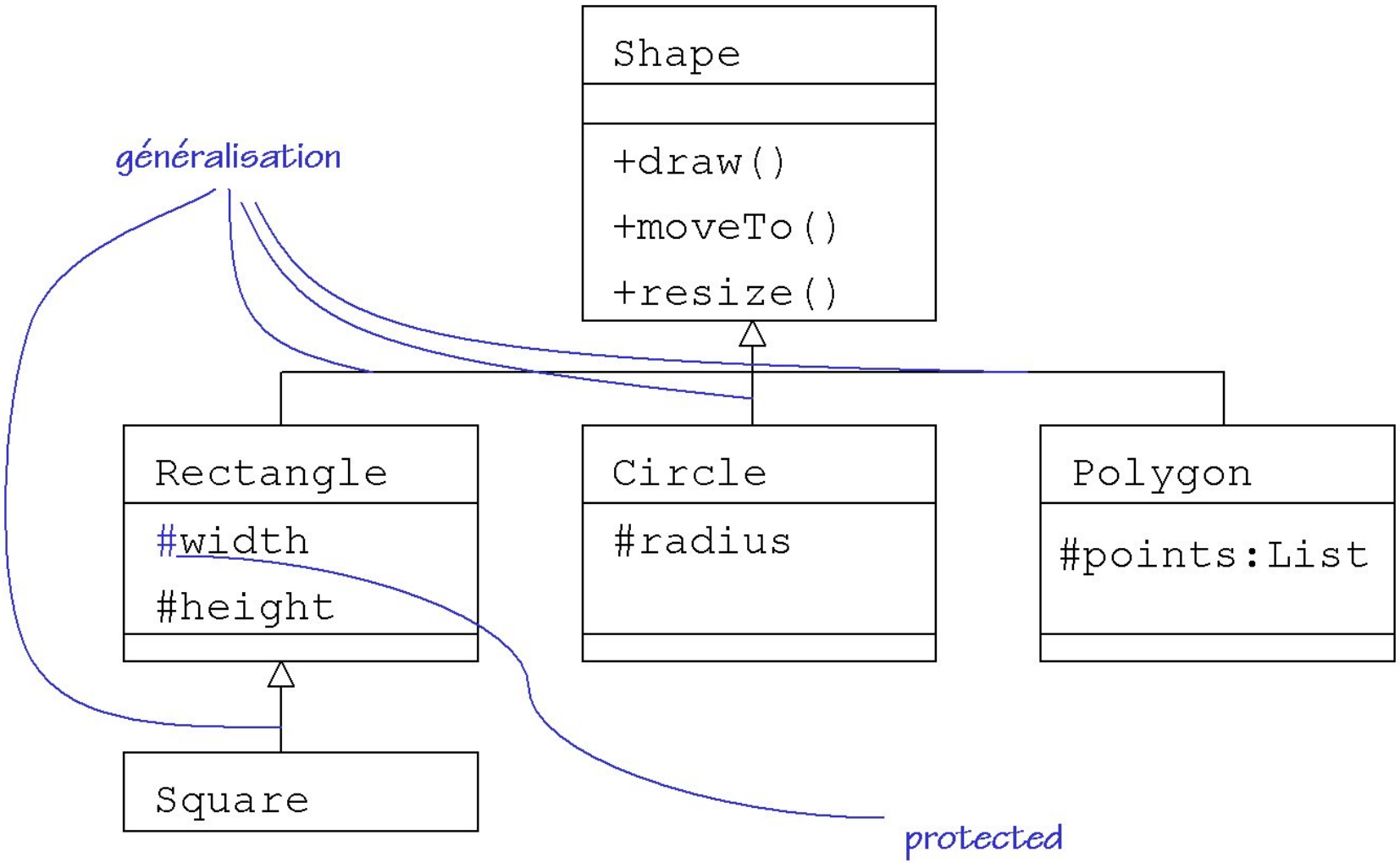
“*friendly*” pour mon club d'amis (mais pas les descendants !)

`public` pour tout le monde

# Encapsulation : principes

- protéger l'accès aux attributs définissant l'état et utiliser des accesseurs et sélecteurs
- les laisser éventuellement accessibles directement pour les sous-classes (ce sont aussi leurs attributs...)
- les attributs "factorisables" peuvent donc être définis comme **`protected`** (et non `private`) et on conserve les accesseurs/sélecteurs pour les autres classes

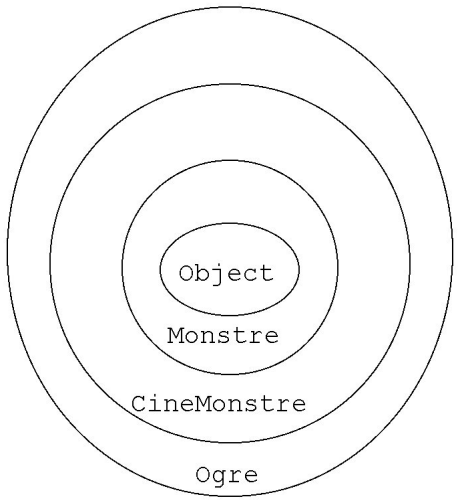
# UML


Shape
+draw()
+moveTo()
+resize()
généralisation
Rectangle
#width
#height
Circle
#radius
Polygon
#points:List
Square
protected

# Plusieurs couches objet...

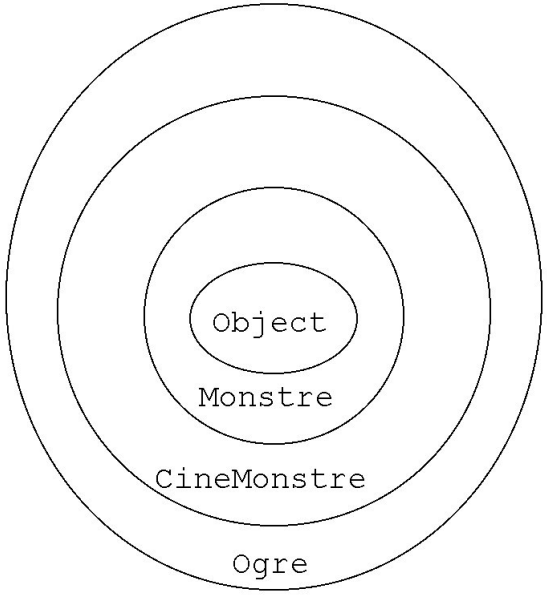
"*les Ogres c'est comme les oignons, ça a des couches*" (Shrek)

```
public class Monstre {...}

public class CineMonstre extends Monstre {...}

public class Ogre extends CineMonstre {...}
```




- l'objet `shrek` est composé d'un **noyau** défini par **`Object`**, **étendu** par une "sur-couche" définie par `Monstre`, **étendue** par une couche `CineMonstre`, **étendue** par une couche `Ogre`.

```
// utilisation  ...
Ogre shrek = new Ogre();
Monstre upcastShrek = shrek;
```




- à chaque couche on peut utiliser tout ce qui est **accessible** aux couches intérieures
- en cas de surcharge, on prend la définition la plus "extérieure"
- lorsque l'on upcast, cela revient à supprimer des couches (cf. `upcastShrek`)
  $\Rightarrow$ on supprime l'accès aux définitions des couches enlevées

# `super` constructeur

- il faut construire les différentes couches
  $\Rightarrow$ utilisation des **constructeurs** pour chaque couche
- pour construire un objet il faut appeler l'un des constructeurs de la super-classe
- on le référence par le mot réservé **`super`** suivi des éventuels paramètres
- cet appel doit être la **première** action dans le constructeur
- peut être implicite dans le cas de l'appel du constructeur sans paramètre

## Construction d'objets de classes héritantes

```
public class Animal {      // le constructeur par défaut de Animal utilise
}                          // implicitement super() de Object
public class Mammifere extends Animal {
  private String nom;
  public Mammifere(String nom) {
    this.nom = nom;        // utilisation implicite de super() avant cette ligne
  }
}
public class Felin extends Mammifere {
  private boolean griffesRetractiles;
  public Felin() {
    super("un felin");     // appel du (seul) constructeur de la super-classe
    this.griffesRetractiles = true;
  }
  public Felin(String nom) {
    super(nom);            // appel du (seul) constructeur de la super-classe
    this.griffesRetractiles = true;
  }
  public Felin(String nom, boolean griffesRetract) {
    super(nom);            // appel du (seul) constructeur de la super-classe
    this.griffesRetractiles = griffesRetract;
  }
  public Felin(boolean griffesRetract) {
    super("un felin");     // appel du (seul) constructeur de la super-classe
    this.griffesRetractiles = griffesRetract;
  }
}
```

# Les étapes de la création d'un objet

1. chargement de la classe (si pas encore fait)
   (et donc chargement de l'éventuelle super-classe (selon même principe))
2. les attributs `static` (avec valeur par défaut)
   ↪ une seule fois au moment du chargement de la classe
3. appel du constructeur de la super-classe,
4. initialisation des attributs ayant une valeur par défaut,
5. exécution du reste du code du constructeur.

```
public class Value {
  public Value(int i) { System.out.println("Value "+i); } }
public class C {
  private static Value v0 = new Value(0);
  public C() { System.out.println("C"); }
}
public class Initialisation extends C {
  private Value v3;
  public Initialisation() {
    System.out.println("Initialisation");
    this.v3 = new Value(3);
  }
  private static final Value v1 = new Value(1);
  private Value v2 = new Value(2);

  public static void main(String[] args) {
    new Initialisation();
    System.out.println("***************");
    new Initialisation();
  }
}
```

```
| trace :
|>java Initialisation
| Value 0
| Value 1
| C
| Value 2
| Initialisation
| Value 3
| ***************
| C
| Value 2
| Initialisation
| Value 3
```

# Mort d'un objet

- a priori il n'y a pas à s'en occuper : **Garbage Collector**
  le GC recycle *si nécessaire* les objets qui ne sont plus utiles, c-à-d qui ne sont plus référencés et libère la mémoire associée.
- pas d'assurance qu'un objet sera collecté
- `finalize` : méthode appelée par le GC (donc **pas** forcément appelée !)
  ↪ permet un traitement spécial lors de la libération par le GC
  ↪ la correction ne doit pas dépendre de l'appel à `finalize`

GC ≠ destruction (cf. C++)
(GC pas systématique et pas spécifié)

GC → uniquement libération de ressources mémoire

# finalize

- peut être nécessaire si de la mémoire a été allouée autrement que par Java (ex : par programme C ou C++ via JNI)
- `finalize()` n'est appelée qu'une unique fois pour un objet...
- GC en deux passes :
  1. détermine les objets qui ne sont plus référencés et appelle `finalize()` pour ces objets
  2. libère effectivement la mémoire

**Méthode :** Si l'on veut un traitement particulier (autre que mémoire) lors de la fin de la vie de l'objet : construire et appeler explicitement une méthode dédiée (pas `finalize()`) (cf. destructeur C++)
(ex : fermer des flux)

## Mort d'un objet

```
public class Value {
  static int cpt = 0;
  private int idx;
  public Value() { this.idx = cpt++; }
  public void finalize() {
    System.out.println(this.idx+ " finalized");
  }
}
public class TestFinalize {
  public static void main(String[] args) {
    for (int i=0;i< Integer.parseInt(args[0]);
             i++) {
      new Value();
    }
  }
} // TestFinalize
```

```
| trace :
| >java TestFinalize 10000
|
| >java TestFinalize 11000
| 0 finalized
| 1 finalized
| 2 finalized
| 3 finalized
| 4 finalized
| 5 finalized
| 6 finalized
| 7 finalized
| 8 finalized
| 9 finalized
| 10 finalized
```

# Incertitude et finalisation

extrait doc API (1.2, idem dans 1.5) :

`java.lang.System` `public static void runFinalization()`

> *Runs the finalization methods of any objects pending finalization.*
>
> *Calling this method **suggests** that the Java Virtual Machine **expends effort toward running** the finalize methods of objects that have been found to be discarded but whose finalize methods have not yet been run. When control returns from the method call, the Java Virtual Machine **has made a best effort** to complete all outstanding finalizations.*