

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 (≠ fainéant !)

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

Programmer c'est investir !

Université Lille 1 - Licence Informatique						1	Université Lille 1 - Licence Informatique						2
Réutiliser	Héritage	protected	Construction	super	Exemples	Création	Réutiliser	Héritage	protected	Construction	super	Exemples	Création
●○○○○	○○○○○○○○○○○○○○○○	○○○○○○○○	○○○○	○○○○	○○	○○○○○	●○○○○	○○○○○○○○○○○○○○○○	○○○○○○○○	○○○○	○○○○	○○	○○○○○

... 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. ImageManipulator et interface Image

... un comportement

(par agrégation/composition)

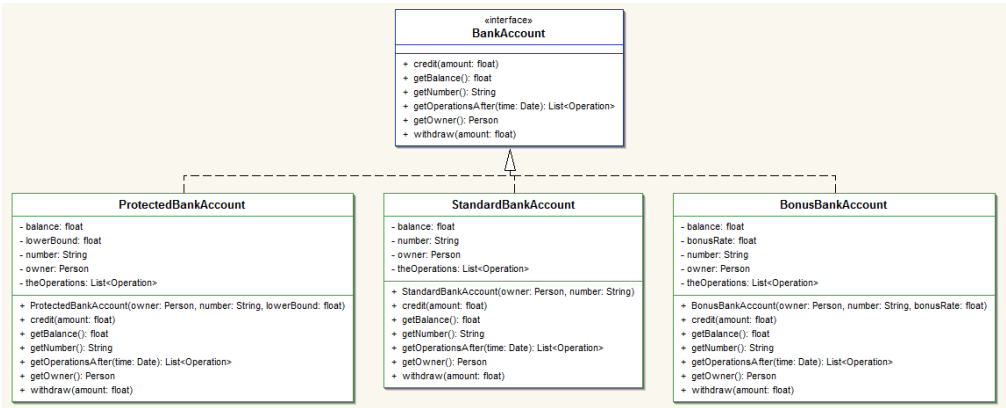
réutilisation partielle et nécessitant une contextualisation

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 Strat1 implements Strategie { ... }
public class Strat2 implements Strategie { ... }
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*
- **réutiliser un comportement** : on n'a pas besoin de réécrire le code des méthodes **mais** on n'a pas de compatibilité de type et donc de polymorphisme



Héritage

Héritage de classe

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

- attributs et codes *répétés* ! \Rightarrow **factorisation de code** possible
- mais credit ou withdraw *différents*...

comment concilier la factorisation de code et les différences ?

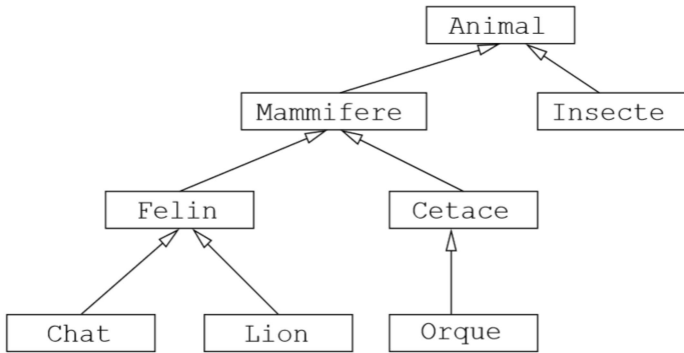
- la classe héritante
 - **recupère tous** les comportements (accessibles) de la classe dont elle hérite,
 - **peut modifier** certains comportements hérités,
 - **peut ajouter** de nouveaux comportements qui lui sont propres.
- les instances de la classe héritante sont également du type de la classe héritée : **polymorphisme**
 - \hookrightarrow on parle de **sous-classe** (donc sous-type)
 - \hookrightarrow 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.

extends

Object : des mystères révélés

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.

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 **directement 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;           // attribut public juste pour illustrer
}

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
... **alors qu'ils existent !**

```
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

extension du comportement

- 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()

Felin ⊂ Mammifere ⊂ Animal

Spécialisation

spécialisation du comportement

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

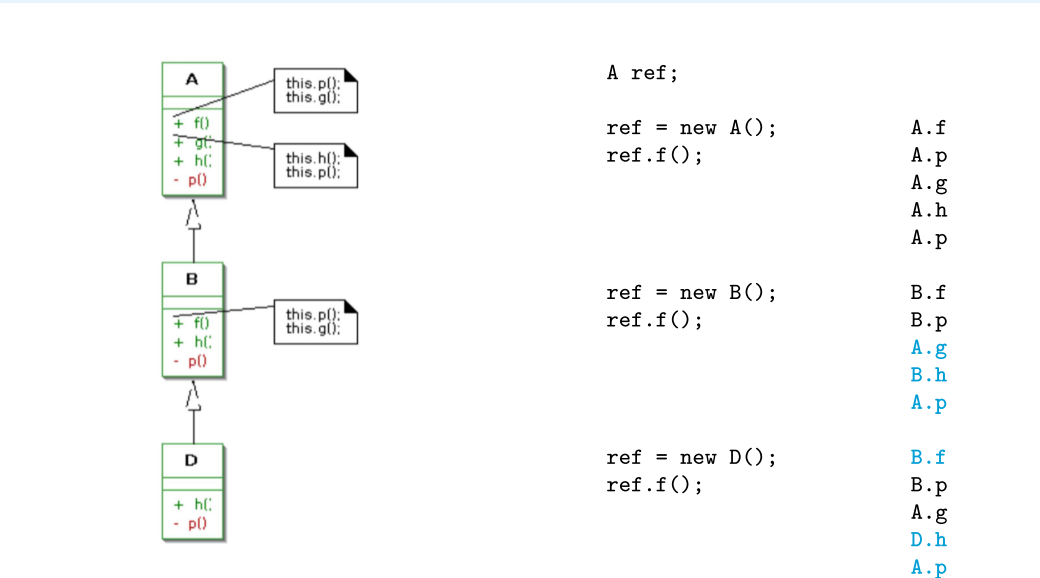
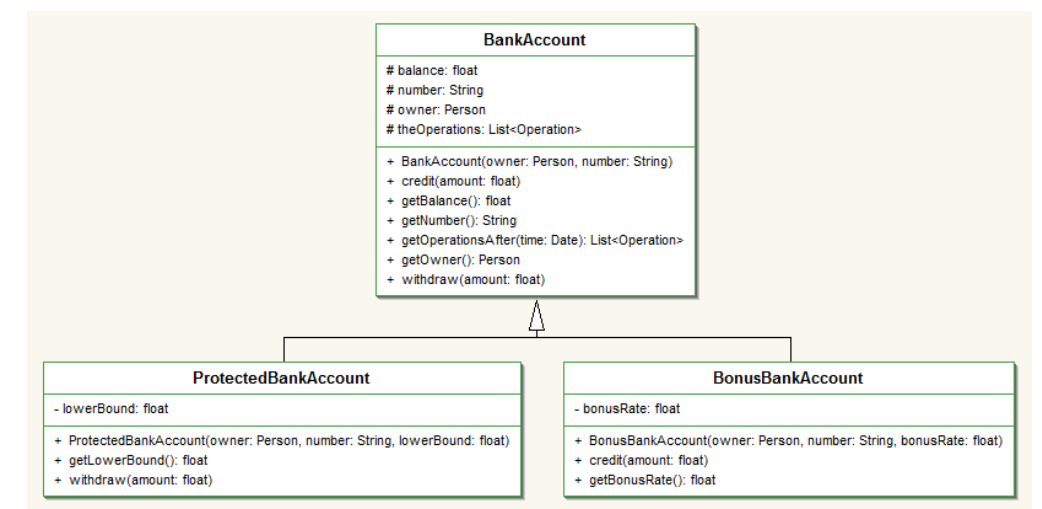
NB : la surcharge permet d'accroître la visibilité d'une méthode pas de la réduire

```
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"
```

Recherche de méthodes (lookup)



NB : méthode p **private** - Méthodes affichent *NomClasse.NomMéthode*

Recherche de méthodes

- **this** est une référence vers l'objet qui reçoit le message : celui qui invoque la méthode

Recherche ("lookup") :

- méthode publique :
 - la recherche commence dans la **classe de l'objet invoquant**
 - si aucune définition de la méthode n'est trouvée, on continue la recherche dans la super-classe
 - et ainsi de suite
- la recherche de la méthode est donc **dynamique**
- si le message porte sur une méthode **privée** de la classe du receveur :
 - on prend la définition de la méthode dans la classe où est réalisée l'envoi de message. Dans ce cas le choix est **statique**.

Attributs : masquage

```
public class Mammifere extends Animal {
    public int nbDePattes = 4;        // public pour illustrer
}
public class Cetace extends Mammifere {
    public int nbDePattes = 0;        // idem
}

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 pour illustrer
}
public class Cetace extends Mammifere {
    public int nbPattes = 0;        // idem
}

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

Moralité

Eviter les surcharges d'attributs ! (sens ?)

protected

Modificateurs d'accès

- 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
    }
}

```

<i>modificateur\accès</i>	classe	classe héritée	même paquetage	autre cas
private	×	—	—	—
protected	×	×	×	—
aucun (" <i>friendly</i> ")	×	—	×	—
public	×	×	×	×

private

protected

"friendly"

public

for my eyes only

pour mes descendants et mes amis (!)

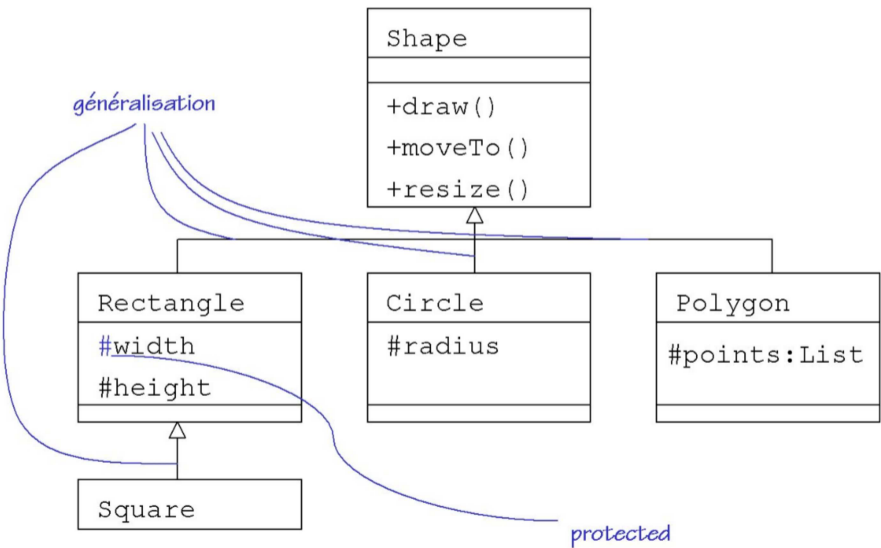
pour mon club d'amis (mais pas les descendants !)

pour tout le monde

Encapsulation : principes

UML

- 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 on conserve les accesseurs/sélecteurs pour les autres classes



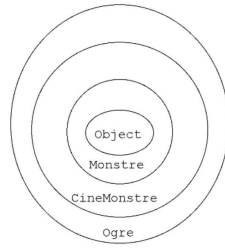
Plusieurs couches objet...

“les Ogres c’est comme les oignons, ça a des couches” (Shrek)

```
public class Monstre {...}

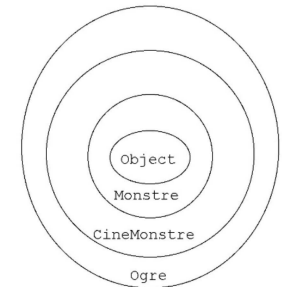
public class CineMonstre extends Monstre {...}

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 l’accès à des couches (cf. upcastShrek)
⇒ on supprime l’accès aux définitions des couches enlevées

Université Lille 1 - Licence Informatique							29	Université Lille 1 - Licence Informatique							30
Réutiliser ○○○○○	Héritage ○○○○○○○○○○○○○○○○	protected ○○○○○○○	Construction ●○○○	super ○○○○	Exemples ○○	Création ○○○○○		Réutiliser ○○○○○	Héritage ○○○○○○○○○○○○○○○○	protected ○○○○○○○	Construction ○○○●	super ○○○○	Exemples ○○	Création ○○○○○	

super constructeur

- il faut construire les différentes couches
⇒ 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

```
public class Animal {    // le constructeur par défaut de Animal utilise
                        // implicitement super() de Object
}

public class Mammifere extends Animal {
    protected String nom;
    public Mammifere(String nom) {
        this.nom = nom;    // utilisation implicite de super() avant cette ligne
    }
}

public class Felin extends Mammifere {
    protected 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;
    }
}
```


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
- **super** est une référence vers l'objet qui reçoit le message ("invoquant")
`super == this`

```
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
```

si dans BankAccount :

```
public void credit(float amount) {
    this.balance = this.balance + amount;
    Date date = Calendar.getInstance().getTime();
    this.theOperations.add(new Operation(amount, date));
}
```

alors dans BonusBankAccount extends BankAccount :

```
public void credit(float amount) {
    this.balance = this.balance + (amount * this.bonusRate);
    Date date = Calendar.getInstance().getTime();
    this.theOperations.add(new Operation(amount, date));
}
```

devient

```
public void credit(float amount) {
    super.credit(amount*this.bonusRate);
}
```

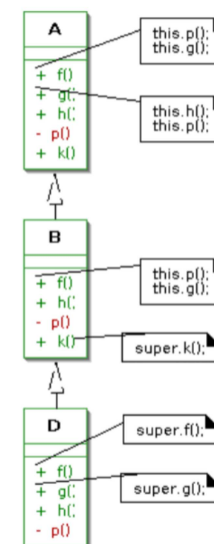
lookup avec super

- **super** est une référence vers l'objet qui reçoit le message ("invoquant")
`super == this`

Mais, recherche de méthode (*lookup*) avec **super** **différente** :

- méthode publique :
 - la recherche commence dans la **super-classe de la classe définissant la méthode** utilisant **super**
 - le processus de chaînage reste ensuite le même
- **super ne fait pas** commencer la recherche de méthode dans la super-classe de l'objet

Recherche avec super



```
A ref;

ref = new B();
ref.k();           B.k
                   A.k

ref = new D();
ref.f();           D.f
                   B.f
                   B.p
                   D.g
                   A.g
                   D.h
                   A.p
                   B.k
                   A.k

ref.k();           A.k
```

this est dynamique, **super** est statique

Créer une classe d'exception

Méthodes surchargées et exceptions

- dans la mesure du possible utiliser les exceptions existantes, sinon
- 1 définir une classe en lui donnant un nom explicite de la forme QuelqueChoseException
- 2 la faire hériter de Exception ou de l'une de ses sous-classes déjà définies

```
public class MatiereNotFoundException extends Exception {
    public MatiereNotFoundException(String msg) {
        super(msg);
    }
}
```

- les exceptions qui héritent de RuntimeException n'ont pas besoin d'être obligatoirement capturées

Lors de la surcharge d'une méthode d'une super-classe, la signature doit être rigoureusement la même, **jusqu'aux** exceptions. Avec cependant la possibilité d'affiner les exceptions levées par la méthodes par des sous-classes des exceptions originales.

```
public class ImmangeableException extends Exception { ... }
public class Animal {
    public void mange(Nourriture n) throws ImmangeableException { ... }
}
public class ViandeImmangeableException extends ImmangeableException { ... }
public class Herbivore extends Animal {
    public void mange(Nourriture n) throws ViandeImmangeableException { ... }
}
// ... utilisation ...
Animal animal = new Animal();
try {
    animal.mange();
}
catch(ViandeImmangeableException e) { ... }
catch(ImmangeableException e) { ... }
```

Attention à l'ordre de capture des exceptions

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

finalize

- 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

- 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)

Incertitude et finalisation

```

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 100000
|
| >java TestFinalize 110000
| 0 finalized
| 1 finalized
| 2 finalized
| 3 finalized
| 4 finalized
| 5 finalized
| 6 finalized
| 7 finalized
| 8 finalized
| 9 finalized
| 10 finalized

```

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.*