

# Programmation Orientée Objet - Chapitre 2 – Héritage : délégation et polymorphisme

Jean-Marie Normand  
Bâtiment E - Bureau 211  
[jean-marie.normand@ec-nantes.fr](mailto:jean-marie.normand@ec-nantes.fr)

# Plan

## 5 Héritage : délégation et polymorphisme

# Notion d'héritage

Dans le cours d'aujourd'hui nous allons :

- voir la notion d'**héritage** en Programmation Orientée Objet
- revenir sur les **droits d'accès** en Java
- réexaminer la notion de **masquage**
- aborder la notion de **redéfinition**
- étudier l'**impact de l'héritage sur les constructeurs**

# Un exemple : jeu de rôle

Comme exemple tout au long de ce cours, nous allons parler du nouveau jeu de rôle : *World of ECN*



# Un exemple : les classes des personnages I

Pour un jeu de rôle, il nous faut des classes représentant les différents types de personnages possibles du jeu.

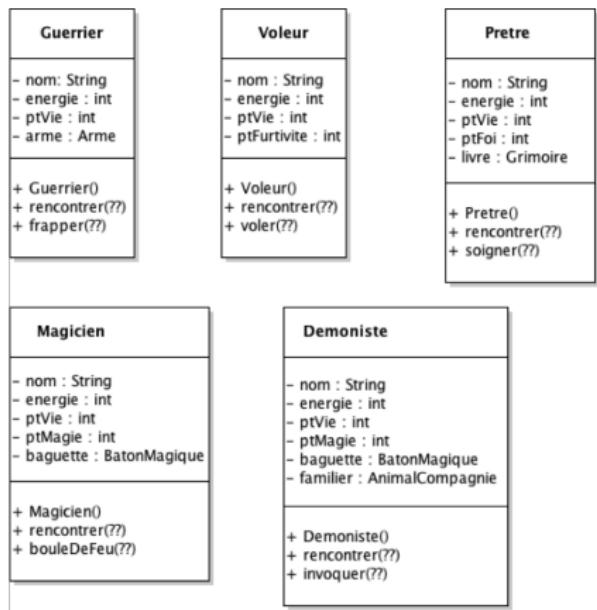


Figure: Les différentes classes de personnages de notre jeu.

# Un exemple : les classes des personnages II

Problèmes sur ces classes :

- différentes classes avec beaucoup d'**attributs** et de **méthodes** en commun !
- beaucoup de code Java dupliqué !
- pour certaines méthodes on ne sait pas **quels types d'arguments** leur passer (ou alors il faudrait une méthode pour chaque type de paramètre)
- si l'on veut modifier les caractéristiques des personnages (p. ex. rajouter un attribut ou une méthode) ⇒ il faut modifier **TOUTES** les classes une par une !

# Un exemple : les classes des personnages II

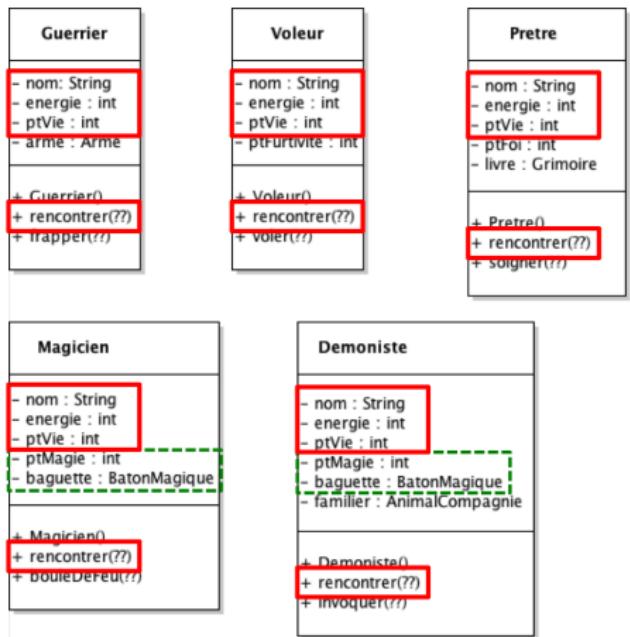


Figure: Les choses communes des différentes classes de personnages de notre jeu.

# Un exemple : les classes des personnages II

Solution : l'héritage !

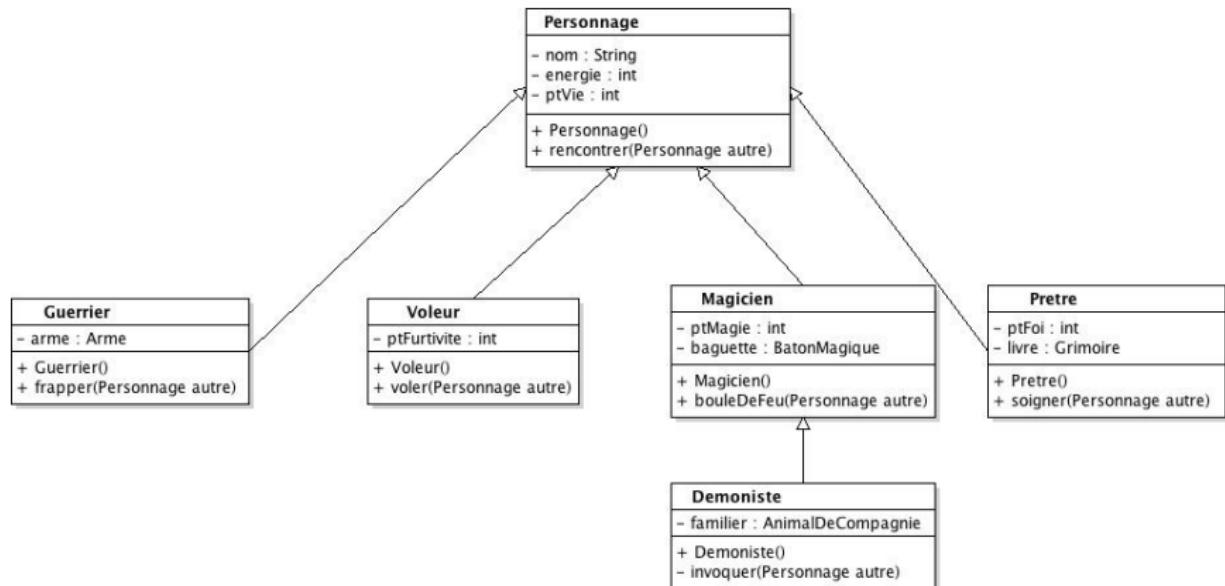
L'héritage permet :

- de regrouper les parties communes (attributs, méthodes) au sein d'une **super-classe** (aussi appelée classe mère)
- de **spécialiser** cette super-classe en **rajoutant des choses plus spécifiques** dans des **sous-classes** (aussi appelées classes filles)

**L'héritage est un aspect fondamental de la programmation orientée objet !**

# Un exemple : les classes des personnages III

Solution avec une super classe **Personnage** et des sous-classes pour chaque type “spécialisé” de personnage



# Héritage I

L'héritage représente la relation “**est-un**” et permet, à partir d'une super-classe générale, de définir des sous-classes plus spécialisées

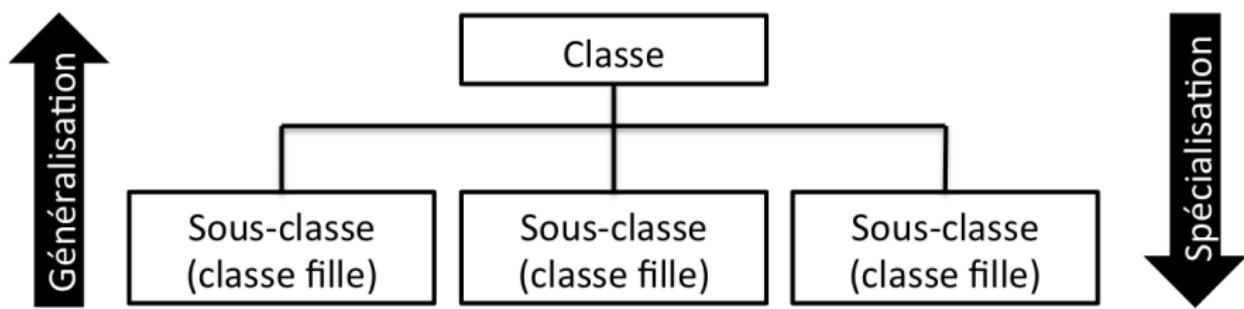


Figure: Vision duale de l'héritage.

# Héritage I

Représentation UML de la relation d'héritage !

En UML, la relation d'héritage est représentée par une **flèche à pointe vide allant de la sous-classe (classe fille) vers la super-classe (classe mère)**

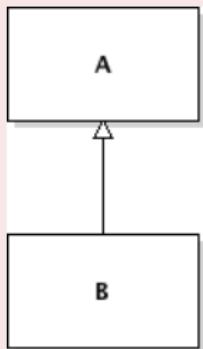


Figure: Représentation de la relation d'héritage en UML.

# Héritage II

Lorsqu'une classe **B** hérite d'une classe **A** :

- le type est hérité : un objet **B** **est** (aussi) **un A**
- **B** va **hériter** de l'ensemble :
  - ▶ des attributs de **A**
  - ▶ des méthodes de **A** (sauf les constructeurs)
- Les attributs et méthodes de **A** sont directement accessibles dans **B** sans avoir à les redéfinir !
- mais, de plus :
  - ▶ des attributs et/ou des méthodes supplémentaires peuvent être défini(e)s dans la sous-classe **B** ( $\Rightarrow$  **enrichissement**)
  - ▶ des méthodes **héritées** de **A** peuvent être redéfinies dans **B** ( $\Rightarrow$  **spécialisation**)

## Héritage : exemple

Lorsqu'une classe **B** (ici **Guerrier**) hérite d'une classe **A** (ici **Personnage**) :

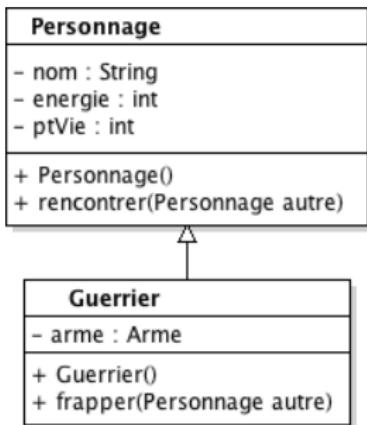
- le type est hérité : un objet **Guerrier est** (aussi) **un Personnage**

```
Personnage p;  
Guerrier g;  
// ...  
p = g;  
// si on suppose avoir une methode : void  
    afficher(Personnage pers) {...}  
afficher(g);
```

# Héritage : exemple

Lorsqu'une classe **B** (ici **Guerrier**) hérite d'une classe **A** (ici **Personnage**) :

- **Guerrier** va hériter de l'ensemble des attributs et des méthodes de la classe **Personnage** (sauf les constructeurs) :



## Explication de code

```

// ... dans une methode main
Guerrier g = new Guerrier(...);
Voleur v = new Voleur(...);
// ...
g.rencontrer(v);
// dans une methode de la classe
// Guerrier :
this.setEnergie(this.getEnergie()-1);
  
```

Figure: Relation d'héritage entre

## Héritage : exemple

Lorsqu'une classe **B** (ici **Guerrier**) hérite d'une classe **A** (ici **Personnage**) :

- des attributs et/ou des méthodes supplémentaires peuvent être défini(e)s dans la sous-classe **Guerrier** : p. ex. l'attribut **arme** ou la méthode **frapper**
- des méthodes héritées de **Personnage** peuvent être redéfinies dans **Guerrier** : p. ex. la méthode **rencontrer(Personnage)**

# Héritage III

L'héritage permet donc :

- d'expliciter des **relations structurelles et sémantiques** entre classes
- de **réduire les redondances de description et de stockage des propriétés** (attributs)

## Attention !

L'héritage est :

- utilisé pour décrire une relation “**est-un**” (“*is-a*”)
- ne doit **jamais** être utilisé pour décrire une relation “**a-un**” ou “**possède-un**” (“*has-a*”) ⇒ ceci est modélisé par l'encapsulation et les attributs d'une classe

# Transitivité de l'héritage

Par transitivité, les instances d'une sous-classe possèdent donc :

- les attributs et méthodes (sauf les constructeurs) de l'ensemble des classes parentes (super-classe, super-super-classe, etc.)

## Enrichissement par héritage

- crée un réseau de dépendances entre les classes
- organise ce réseau en une structure arborescente où chacun des nœuds hérite des attributs et méthodes de l'ensemble des nœuds du chemin remontant jusqu'à la racine
- ce réseau arborescent définit une **hiérarchie de classes**

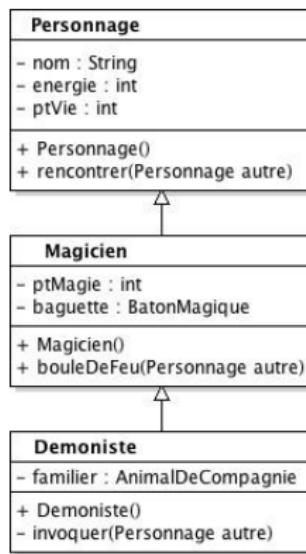


Figure: Transitivité de l'héritage

# Super-classe, Sous-classe

Une **super-classe** :

- est une classe “parente”, “mère”
- peut avoir plusieurs sous-classes
- déclare des attributs/méthodes communs à toutes ses sous-classes

# Super-classe, Sous-classe

Une **super-classe** :

- est une classe “parente”, “mère”
- peut avoir plusieurs sous-classes
- déclare des attributs/méthodes communs à toutes ses sous-classes

Une **sous-classe** :

- est une classe “enfant”, “fille”
- étend, hérite d'**une seule** super-classe
- hérite des **attributs**, des **méthodes** et du **type** de la super-classe

# Super-classe, Sous-classe

Une **super-classe** :

- est une classe “parente”, “mère”
- peut avoir plusieurs sous-classes
- déclare des attributs/méthodes communs à toutes ses sous-classes

Une **sous-classe** :

- est une classe “enfant”, “fille”
- étend, hérite d'**une seule** super-classe
- hérite des **attributs**, des **méthodes** et du **type** de la super-classe

Un attribut/une méthode héritée peut s'utiliser comme si il/elle était déclarée dans la sous-classe au lieu de la super classe :

- suivant les droits d'accès utilisés (voir plus loin)
- on évite la **duplication de code** (c'est la relation “est-un”)

# Mini Quiz



Votons à mains levées !

Soient les classes **Oiseau**,  
**Animal**, **Chat**, **Chien** ...

- ➊ **Chien** est une super-classe de **Chat**.
- ➋ **Chat** est une sous-classe de **Chien**.
- ➌ **Oiseau** est une super-classe d'**Animal**.
- ➍ **Animal** est une super-classe de toutes les autres.

Soient les classes  
**QuatreQuatre**, **Vehicule**,  
**Decapotable**, **Avion**, **Moto**,  
**Voiture** ...

- ➊ **Vehicule** est la classe la plus haute dans la hiérarchie de classes.
- ➋ **Avion** et **Moto** sont au même niveau de la hiérarchie de classes.
- ➌ **Voiture** est super-classe de **Decapotable**.
- ➍ **QuatreQuatre** est super-classe de **Voiture**.
- ➎ **Decapotable** est une sous-classe de **Vehicule**.

# Héritage en Java

Déclaration d'une sous-classe en Java :

```
public class NomSousClasse extends NomSuperClass {
/* Attributs et des méthodes spécifiques à la sous-classe */
}
```

## Sous-classe en Java

```
/*
 * Classe representant un guerrier
 Guerrier.java (on suppose la classe
 'Arme' ecrive)
 * @author jmnormand
 * @version 1.0
 */
public class Guerrier extends Personnage {
    // Attributs de la classe
    /**
     * Arme du personnage
     */
    private Arme arme;
    // ...
}
```

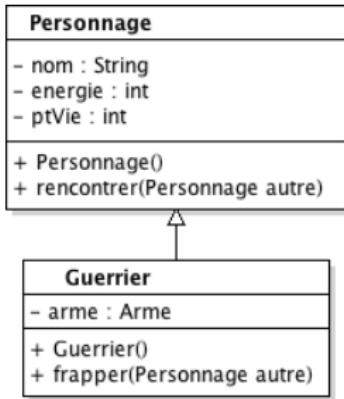


Figure: Sous-classe  
Guerrier de la classe  
Personnage

# Rappels sur les droits d'accès

Jusqu'à présent, nous avons que l'accès aux membres (attributs et méthodes) d'une classe pouvait être :

- **public** : visibilité et accessibilité totale à l'intérieur et à l'extérieur de la classe (mot clé **public**)
- **privé** : visibilité et accessibilité uniquement à l'intérieur de la classe (mot clé **private**)
- **par défaut** (aucun modificateur) : visibilité depuis toutes les classes du même paquetage (voir slides sur les paquetages sur le serveur pédagogique)

# Conséquence sur notre exemple

## Accès aux membres hérités

La classe **Guerrier** hérite de la classe **Personnage** :

- **Guerrier** possède bien l'attribut **ptVie**
- **MAIS** ne peut pas l'utiliser librement !

Attention aux flèches représentant les relations d'héritage UML !

Les schémas des slides 21, 25, 26, 27, 28 et 59 ayant été réalisés avec PowerPoint, la flèche représentant l'héritage **N'EST PAS CORRECTE**. Le standard UML

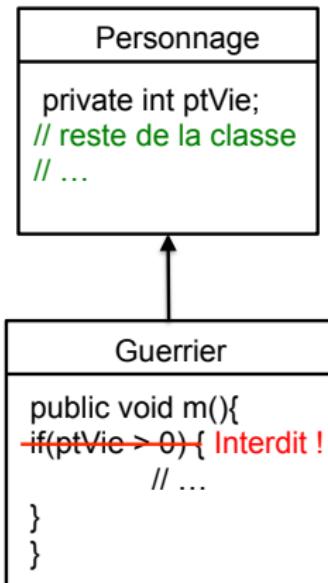


Figure: Accès interdit aux membres privés (ici l'attribut

# Droits d'accès protégé : protected !

Un troisième droit d'accès permet de gérer la visibilité et l'accessibilité des membres (attributs et méthodes) au sein d'une hiérarchie de classes :

- **protégé** : assure la visibilité et l'accessibilité des membres d'une classe dans les sous-classes de sa descendance (et aussi dans les autres classes du même paquetage) Le mot clé associé est : **protected**

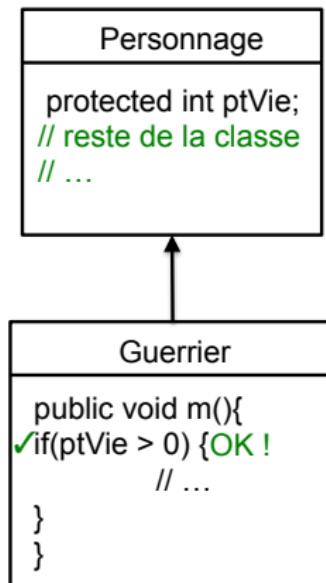


Figure: Accès autorisé aux membres protégés.

## Droits d'accès protégé : `protected` II

- Une sous-classe **n'a pas accès** aux membres (attributs et méthodes) **privés** hérités des super-classes
  - ▶ ⇒ utilisation des accesseurs/modificateurs (*getters/setters*) prévus dans les super-classes
- si une super-classe veut donner le droit d'accès à ses sous-classes à un attribut : elle doit le déclarer en **protected** et non en **private**

### Attention !

La définition d'attributs protégés nuit à une bonne encapsulation ! C'est d'autant plus vrai en Java car les membres protégés sont accessibles par toutes les autres classes d'un même paquetage ! ⇒ les attributs protégés sont peu recommandés en Java !

- Le niveau d'accès protégé correspond à une **extension du niveau privé** permettant l'accès aux membres par les

# Bilan sur les droits d'accès I

- Membres **publics** : accessibles pour les **programmeurs-utilisateurs** de la classe
- Membres **protégés** : accessibles pour les **programmeurs d'extension** de la classe (via l'héritage) ou travaillant dans le même paquetage
- Membres **privés** : définis par le **programmeur de la classe**, définit la structure interne de la classe. Il peut les modifier sans répercussions pour les autres programmeurs et/ou pour les utilisateurs de la classe

# Bilan sur les droits d'accès II

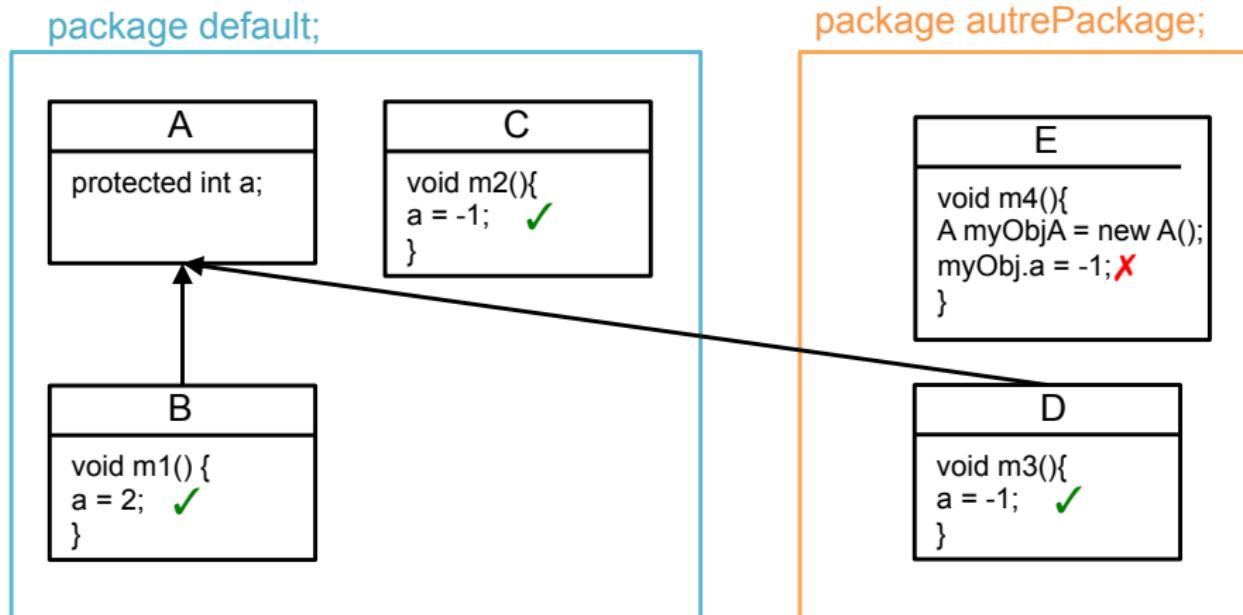


Figure: Bilan sur les droits d'accès : protégé.

# Bilan sur les droits d'accès III

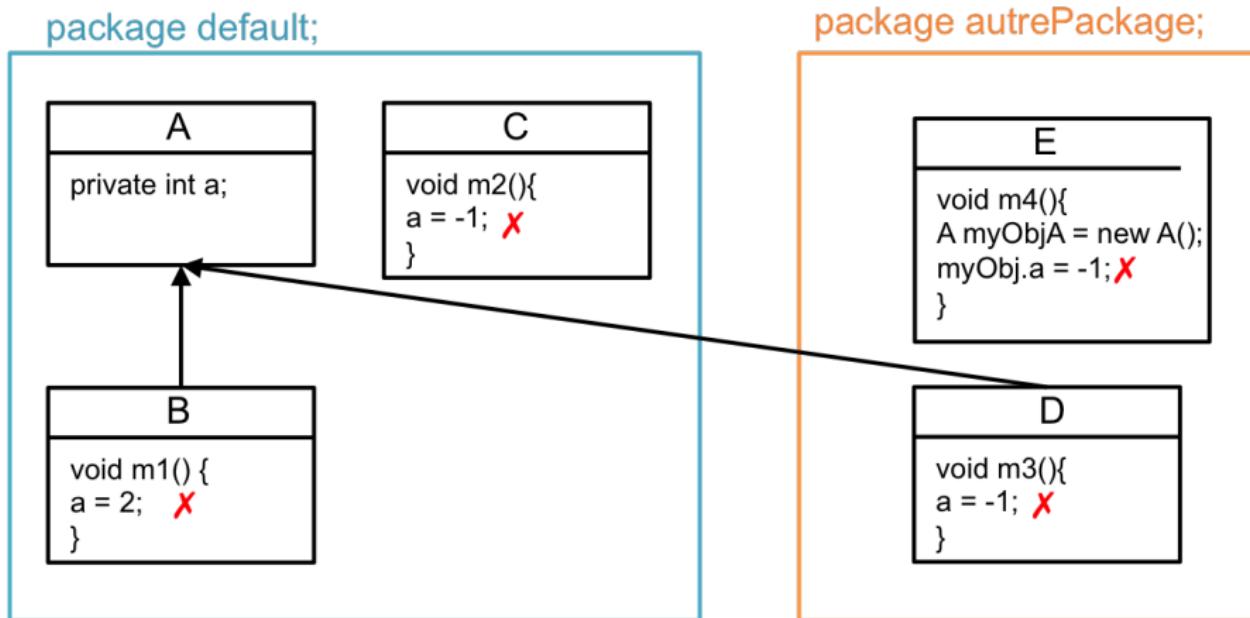


Figure: Bilan sur les droits d'accès : privé.

# Bilan sur les droits d'accès IV

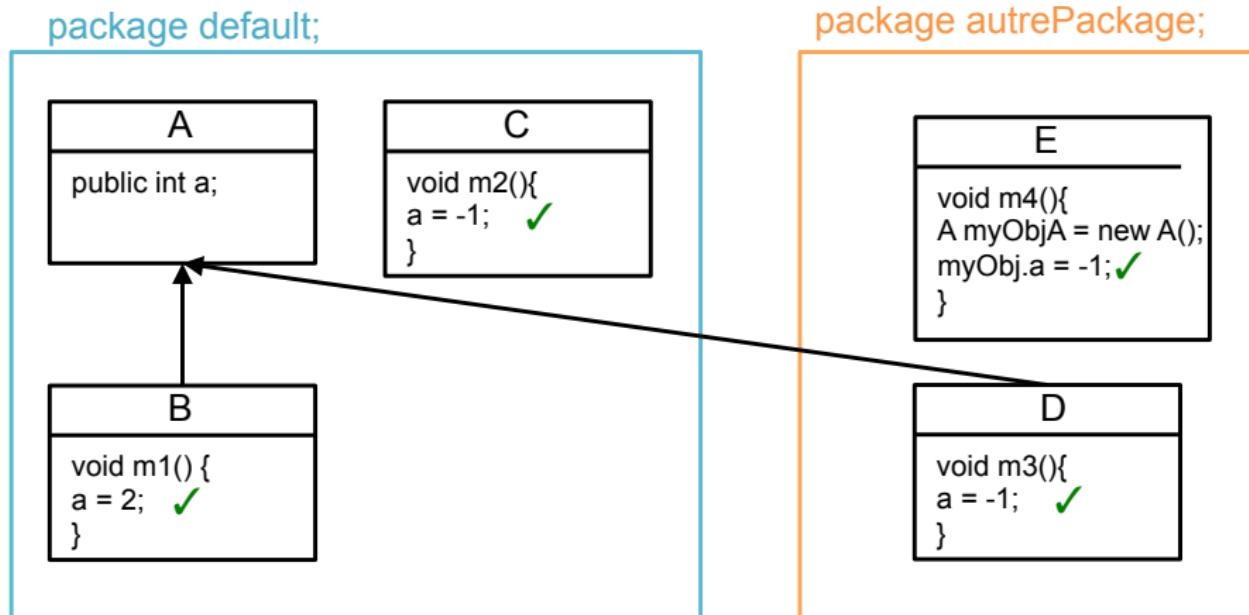


Figure: Bilan sur les droits d'accès : public.

# Bilan sur les droits d'accès V

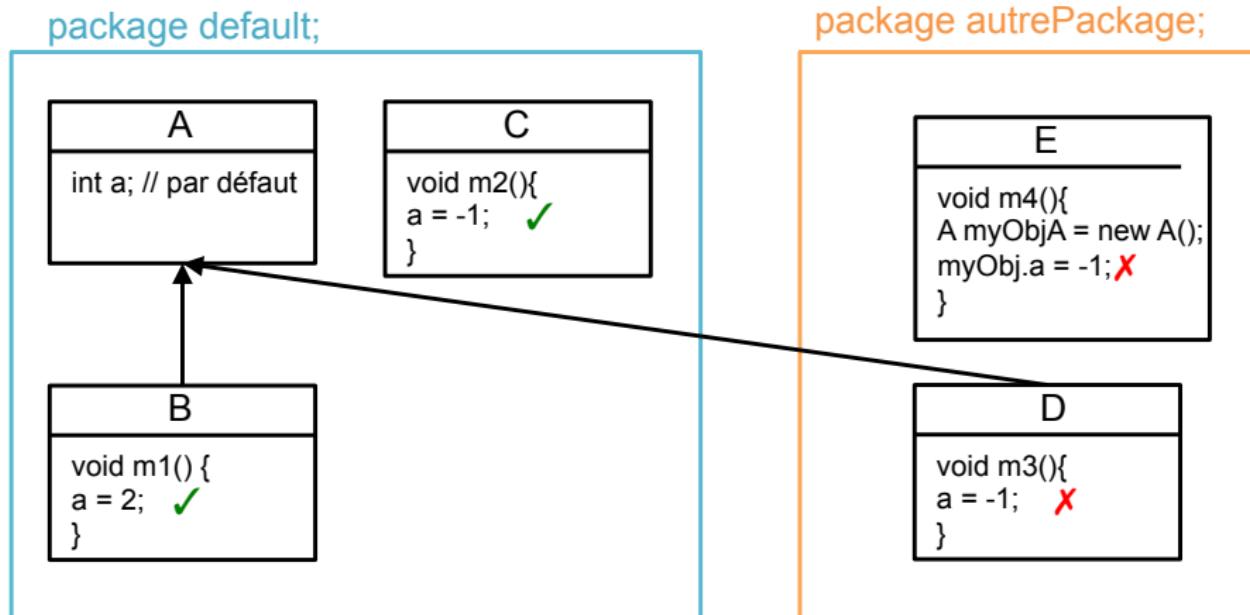


Figure: Bilan sur les droits d'accès : défaut.

# Conséquences de l'héritage I

Nous venons de voir que grâce à l'héritage, les sous-classes possèdent un accès à tous les membres (attributs et méthodes) définis dans la super-classe.

P. ex. dans la hiérarchie de classes suivante, toutes les classes spécialisées de Personnages héritent de la méthode **rencontrer** : elles peuvent donc l'utiliser directement.

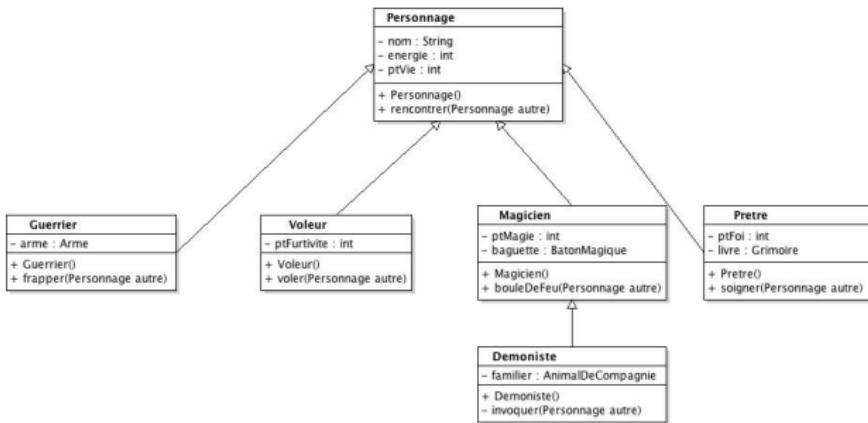


Figure: Hiérarchie des classes de notre jeu avec héritage.

## Conséquences de l'héritage II

Cela suppose que **TOUTES** les classes possèdent la même version de **rencontrer** (celle de **Personnage**) : le comportement de nos personnages sera le même dans toutes les classes !

Et si on ne veut pas avoir partout la même version de cette méthode ?

Imaginons que toutes les classes veulent :

- saluer le personnage qu'elles rencontrent
- **SAUF** les guerriers qui veulent le frapper !

Dans ce cas on aurait :

- pour les personnages non-guerriers :

```
public void rencontrer(Personnage autreP){ saluer(autreP); }
```

- pour les guerriers :

```
public void rencontrer(Personnage autreP){ frapper(autreP); }
```

## Conséquences de l'héritage II

Cela suppose que **TOUTES** les classes possèdent la même version de **rencontrer** (celle de **Personnage**) : le comportement de nos personnages sera le même dans toutes les classes !

Et si on ne veut pas avoir partout la même version de cette méthode ?

Imaginons que toutes les classes veulent :

- saluer le personnage qu'elles rencontrent
- **SAUF** les guerriers qui veulent le frapper !

Dans ce cas on aurait :

Doit-on repenser toute la hiérarchie ?

Non ! Il suffit simplement d'ajouter une nouvelle méthode **rencontrer** dans la classe **Guerrier** qui lui sera spécifique ⇒ c'est la notion de **redéfinition** de méthode en Java

# Exemple de redéfinition de méthode

Nouveau diagramme de classe :

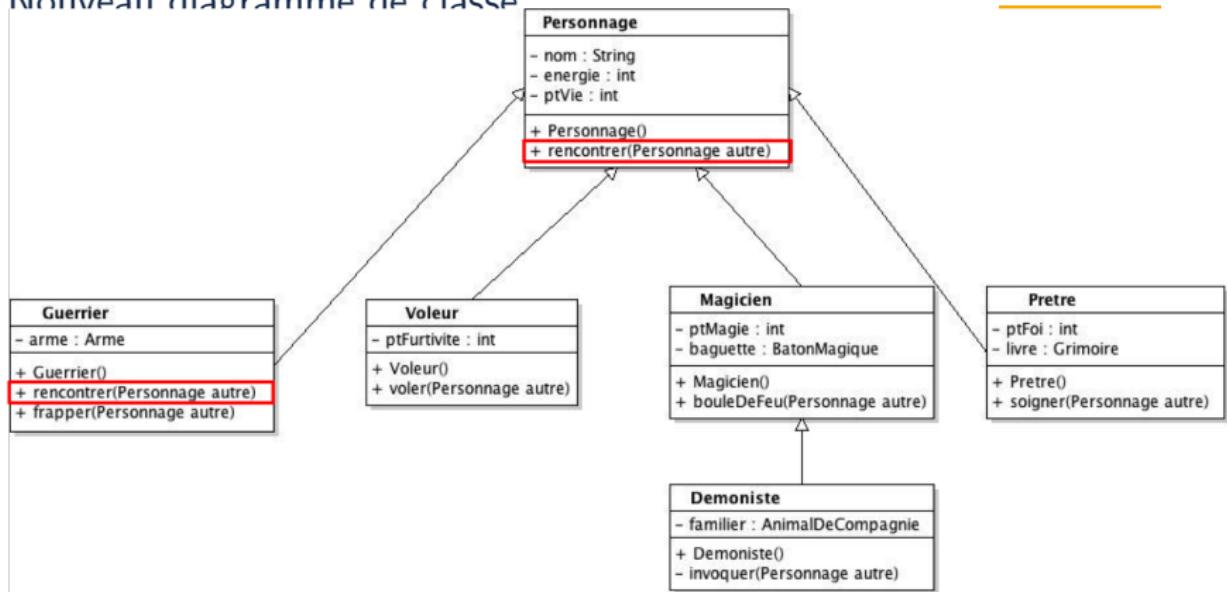


Figure: Nouvelle hiérarchie des classes de notre jeu avec redéfinition d'une méthode.

# Masquage/redéfinition dans une hiérarchie de classes

## Maquage/Redéfinition

Masquage : pour les attributs ("shadowing")

Redéfinition : pour les méthodes ("overriding")

- Masquage : un identificateur qui en cache un autre (défini plus haut dans la hiérarchie de classes)
- Redéfinition : une méthode déjà définie dans une super-classe est redéfinie dans une sous-classe (attention à ne pas faire de surcharge !) ⇒ **l'entête de la méthode redéfinie doit être strictement identique à celui de la super-classe !**
- Différents cas de figures dans une hiérarchie de classes :
  - ▶ même nom d'attribut et/ou de méthode utilisé(e) sur plusieurs niveaux de la hiérarchie
  - ▶ peu courant (et peu recommandé) pour les attributs
  - ▶ très courant et très pratique pour les méthodes !

# Masquage dans une hiérarchie de classes

## Classe A

```
// Une classe 'jouet' avec un attribut unique
public class A {
    private int attrib;
}
```

## Classe B

```
// Une autre classe 'jouet' avec un attribut unique
public class B extends A {
    private int attrib;
}
```

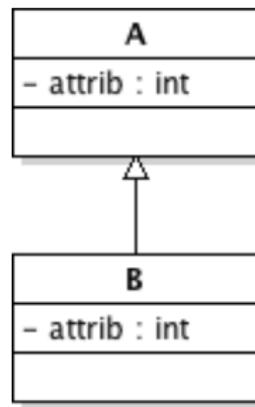


Figure: Masquage d'un attribut de la classe A dans la classe B.

# Masquage dans une hiérarchie de classes II

L'attribut **attrib** de B **masque** celui de A

- un objet de la classe B aura donc **deux** attributs **attrib** !
  - ▶ le sien (i.e. celui de B)
  - ▶ celui de la classe A
- si dans une méthode de la classe B on utilise l'attribut **attrib**
  - ▶ c'est celui de la classe B **qui est utilisé**
  - ▶ et non A

# Redéfinition dans une hiérarchie de classes

## Classe Personnage

```
// ...
public void rencontrer(Personnage autreP) {
    saluer(autreP);
}
// ...
```

## Classe Guerrier

```
// ...
public void rencontrer(Personnage autreP) {
    frapper(autreP);
}
// ...
```

## Personnage

|                                |
|--------------------------------|
| - nom : String                 |
| - energie : int                |
| - ptVie : int                  |
| + Personnage()                 |
| + rencontrer(Personnage autre) |

## Guerrier

|                                |
|--------------------------------|
| - arme : Arme                  |
| + Guerrier()                   |
| + rencontrer(Personnage autre) |
| + frapper(Personnage autre)    |

Figure: Redéfinition d'une méthode dans la classe Guerrier.

# Redéfinition dans une hiérarchie de classes II

La méthode **rencontrer** de **Guerrier** **redéfinit** celle de **Personnage**

- un objet de type **Guerrier** n'utilisera donc **JAMAIS** la méthode **rencontrer** de la classe **Personnage**
- vocabulaire orienté objet (OO) :
  - ▶ méthode héritée = **méthode par défaut**
  - ▶ méthode qui redéfinit la méthode héritée = **méthode spécialisée** (ou **méthode redéfinie**)

# Accès à une méthode redéfinie ou à un attribut masqué

- il est parfois nécessaire d'accéder à un attribut masqué **et/ou à** une méthode redéfinie au sein d'une hiérarchie de classes
- comment faire ?

## Exemple avec nos personnages

Les **Guerriers** saluent maintenant le **Personnage** rencontré (comportement de toutes les autres classes de la hiérarchie) avant de les frapper !

Modifications dans le code :

- Personnage non-**Guerrier** :
  - ▶ *méthode générale* (**rencontrer** de la classe **Personnage**)
- Personnage **Guerrier** :
  - ▶ *méthode spécialisée* (**rencontrer** de la classe **Guerrier**)
  - ▶ **appel** à la méthode générale **dépends** de la méthode spécialisée ⇒ le code étant déjà présent dans la classe **Personnage** il est inutile

# Accès à une méthode redéfinie ou à un attribut masqué II

En Java, pour accéder à un attribut masqué ou à une méthode redéfinie de la super-classe :

- on utilise le mot clé `super`
- syntaxe : `super.nomMéthode` ou `super.nomAttribut`

## Classe Guerrier

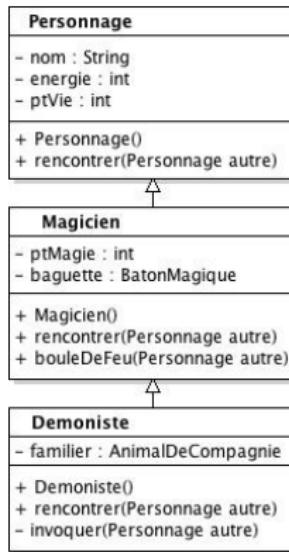
```
public class Guerrier extends Personnage {  
    // ...  
    /**  
     * Une methode qui permet de rencontrer un autre personnage.  
     * Redefinition de la methode 'rencontrer' de la classe 'Personnage'  
     * @param autreP le personnage que l'on va rencontrer  
     */  
    public void rencontrer(Personnage autreP) {  
        super.rencontrer(autreP); // par politesse  
        frapper(autreP);  
    }  
    // ...  
}
```

# Précisions sur l'utilisation de super en Java

En Java, il est **interdit** d'enchaîner les appels à **super**

## Dans la classe Demoniste

```
// Fichier Demoniste.java ...
public void rencontrer(Personnage
    autreP) {
    super.super.rencontrer(autreP);
    // Erreur de compilation !!!
    // C'est interdit en Java
}
// ...
```



**Figure:** Redéfinition d'une méthode à différents niveaux dans la hiérarchie de classes.

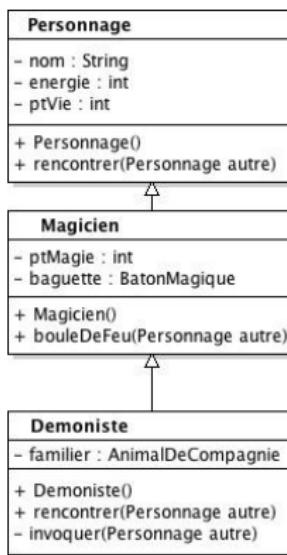
# Précisions sur l'utilisation de super en Java

## II

Attention à l'emplacement des redéfinitions dans la hiérarchie de classes !

### Dans la classe Demoniste

```
// Fichier Demoniste.java ...
public void rencontrer(Personnage
    autreP) {
    super.rencontrer(autreP); // OK !
    // C'est la méthode de Personnage
    // qui est appelée
}
// ...
```



# Compléments sur la redéfinition de méthodes

## Rappel sur la redéfinition de méthode

Il est possible de redéfinir une méthode si l'on respecte les conditions suivantes, la méthode redéfinie doit :

- **porter le même nom** que la méthode qu'elle remplace
- **avoir le même type de retour** que la méthode qu'elle remplace
- **avoir la même liste et types de paramètres** que la méthode qu'elle remplace

## Appel à une méthode redéfinie

Nous venons de voir qu'il est possible (**mais pas du tout obligatoire**) d'appeler la méthode de sa **classe mère** que l'on vient de redéfinir :

- en utilisant le mot clé **super**

## Priorité des méthodes redéfinies sur les méthodes héritées

Lorsqu'une instance de la sous-classe appelle la méthode : c'est la méthode redéfinie qui est appelée en priorité sur la méthode de la super-classe. Ce mécanisme s'appelle la **résolution statique des messages**

# Résolution statique des messages

Afin d'expliquer ce mécanisme, nous allons étudier trois classes **A**, **B** et **C** :

- **A** définit 3 méthodes : **X()**, **Y()** et **Z()**
- la classe **B** hérite de **A** et redéfinit **X()**
- la classe **C** hérite de **B** et redéfinit **X()** et **Y()**

Nous allons regarder ce qui se passe lorsque trois instances **a**, **b** et **c** (respectivement de **A**, **B** et **C**) appellent ces trois méthodes.

# Résolution statique des messages II

| Instance / Méthode | X() | Y() | Z() |
|--------------------|-----|-----|-----|
| a                  | ?   | ?   | ?   |
| b                  | ?   | ?   | ?   |
| c                  | ?   | ?   | ?   |

Table: Méthode effectivement appelée en fonction de l'instance.

Dans la table :

- A signifie que c'est le code de la classe A qui est effectivement appelé
- B signifie que c'est le code de la classe B qui est effectivement appelé
- C signifie que c'est le code de la classe C qui est effectivement appelé

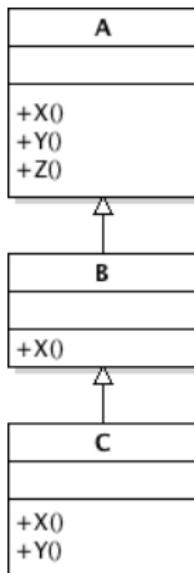


Figure: Hiérarchie des classes A, B et C.

# Résolution statique des messages II

| Instance / Méthode | X() | Y() | Z() |
|--------------------|-----|-----|-----|
| a                  | A   | A   | A   |
| b                  | ?   | ?   | ?   |
| c                  | ?   | ?   | ?   |

Table: Méthode effectivement appelée en fonction de l'instance.

Dans la table :

- A signifie que c'est le code de la classe A qui est effectivement appelé
- B signifie que c'est le code de la classe B qui est effectivement appelé
- C signifie que c'est le code de la classe C qui est effectivement appelé

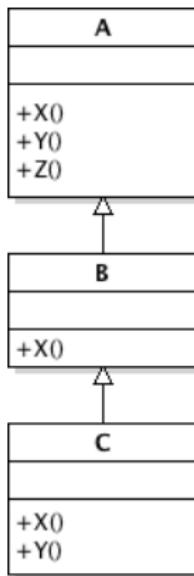


Figure: Hiérarchie des classes A, B et C.

# Résolution statique des messages II

| Instance / Méthode | X() | Y() | Z() |
|--------------------|-----|-----|-----|
| a                  | A   | A   | A   |
| b                  | B   | A   | A   |
| c                  | ?   | ?   | ?   |

Table: Méthode effectivement appelée en fonction de l'instance.

Dans la table :

- A signifie que c'est le code de la classe A qui est effectivement appelé
- B signifie que c'est le code de la classe B qui est effectivement appelé
- C signifie que c'est le code de la classe C qui est effectivement appelé

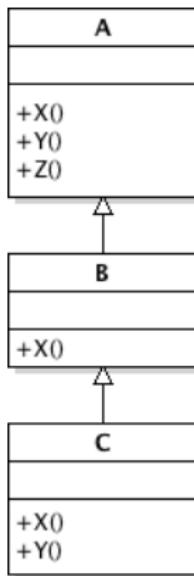


Figure: Hiérarchie des classes A, B et C.

# Résolution statique des messages II

| Instance / Méthode | X() | Y() | Z() |
|--------------------|-----|-----|-----|
| a                  | A   | A   | A   |
| b                  | B   | A   | A   |
| c                  | C   | C   | A   |

Table: Méthode effectivement appelée en fonction de l'instance.

Dans la table :

- A signifie que c'est le code de la classe A qui est effectivement appelé
- B signifie que c'est le code de la classe B qui est effectivement appelé
- C signifie que c'est le code de la classe C qui est effectivement appelé

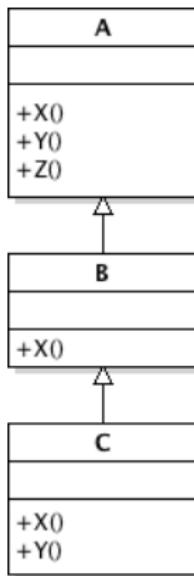


Figure: Hiérarchie des classes A, B et C.

# Héritage et héritage de type

Prenons un exemple de notre jeu *World of ECN* :

## World of ECN

```
// Fichier TestWoECN.java
public static void main(String[] args) {
    // Creation du joueur
    Personnage leJoueur = new Personnage(...);
    // et des PnJ
    Personnage[] pnj = new Personnage[5];
    pnj[0] = new Guerrier(...); // a t'on le droit ??
    pnj[1] = new Voleur(...);
    pnj[2] = new Magicien(...);
    pnj[3] = new Demoniste(...);
    pnj[4] = new Pretre(...);
    // un peu d'action
    for(int i=0; i<pnj.length; i++) {
        pnj[i].rencontrer(leJoueur);
    }
}
```

# Héritage et héritage de type

## World of ECN

```
// Fichier TestWoECN.java
public static void main(String[] args) {
    // Creation du joueur
    Personnage leJoueur = new Personnage(...);
    // et des PNJ
    Personnage[] pnj = new Personnage[5];
    pnj[0] = new Guerrier(...); // a t'on le droit ??
    // ...
}
```

## Question

Peut-on mettre un **Guerrier**, un **Voleur**, etc. dans un tableau d'objets de type **Personnage** ?

# Rappel Héritage de type et Surclassement

Dans une hiérarchie de classe :

- un objet d'une sous-classe hérite du type de sa super-classe
- l'héritage est transitif !
- un objet peut donc avoir **plusieurs types** !

## Surclassement

L'héritage de type implique donc qu'un objet de type **Guerrier** (aussi valable pour nos autres classes de personnages spécialisées) peut être vu comme un objet de type **Personnage** !

Ce mécanisme s'appelle le **surclassement**, est supporté par tous les langages à objets et fonctionne sur plusieurs niveaux d'héritage. Ainsi un objet de type **Démoniste** peut également être vu comme un objet de type **Personnage** !

# Rappel Héritage de type et Surclassement II

## Conséquence immédiate du surclassement :

Les fonctionnalités de l'instance qui vient d'être surclassée seront donc limitées à celle de la classe vers laquelle on vient de la surclasser.

Par exemple, un objet de type **Guerrier** surklassé en **Personnage** ne pourra pas appeler la méthode **frapper()**

## Exemple de surclassements en Java

```
// il est possible de surclasser comme ceci
Personnage pnjG = new Guerrier(...);
// ou bien comme cela
Guerrier monG = new Guerrier(...);
// (transtypage ou cast)
Personnage monP = (Personnage)monG;
// plus de details en TD/TP
```

# Opérateur instanceof

L'opérateur instanceof :

Permet de connaître le type d'une instance !

Exemple d'utilisation de instanceof en Java

```
Guerrier monG = new Guerrier(...);
Demoniste monD = new Demoniste(...);
Magicien monM = new Magicien(...);
// plus loin dans le code
if(monG instanceof Guerrier){ ... /* est vrai */ }
if(monD instanceof Demoniste){ ... /* est vrai */ }
if(monD instanceof Magicien){ ... /* est vrai */ }
if(monM instanceof Demoniste){ ... /* est faux */ }

        devoir écrire comme :monM =(Magicien)monD
monM = monD; // NB : on ne peut pas écrire monD = monM. Pourquoi?
if(monM instanceof Demoniste){ ... /* est vrai */ } pas assi d'information
if(monM instanceof Magicien){ ... /* est vrai */ }
```

# Héritage, surclassement et choix de la méthode à exécuter

Que fait le code suivant ?

```
// ...
Personnage pnjG = new Guerrier(...);
pnjG.rencontrer(leJoueur);
// ...
```

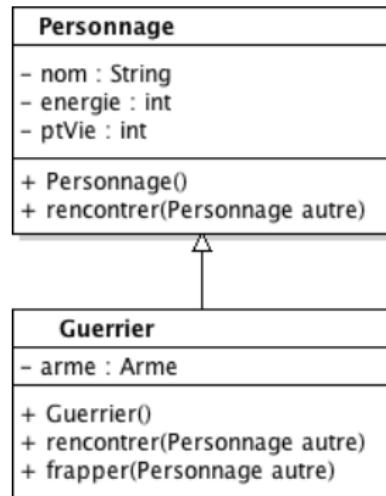


Figure: Redéfinition d'une méthode dans la classe **Guerrier**.

# Héritage, surclassement et choix de la méthode à exécuter II

Que fait le code suivant ?

```
// ...
Personnage pnjG = new Guerrier(...);
pnjG.rencontrer(leJoueur);
// ...
```

Il existe ici **deux possibilités** :

## ① Résolution *statique* des liens :

- ▶ le **type apparent** (de la variable) est déterminant
- ▶ **pnjG** est **déclaré** comme une variable de type **Personnage**
- ▶ méthode exécutée : **rencontrer()** de la classe **Personnage** ! (on salue le joueur)

# Héritage, surclassement et choix de la méthode à exécuter III

Que fait le code suivant ?

```
// ...
Personnage pnjG = new Guerrier(...);
pnjG.rencontrer(leJoueur);
// ...
```

Il existe ici deux possibilités :

## 2 Résolution dynamique des liens :

- ▶ le **type effectif** (ou **type réel**) (i.e. celui de l'objet effectivement stocké dans la variable) est déterminant
- ▶ la variable **pnjG** contient la référence à un objet **créé** comme un objet de type **Guerrier**
- ▶ méthode exécutée : **rencontrer()** de la classe **Guerrier** ! (on salue le joueur puis on le frappe)

# Résolution dynamique des liens

Pour nous :

Java met en œuvre la **résolution dynamique des liens** ! C'est donc le type **effectif** (ou **réel**) et non le type apparent qui est pris en compte.

## En conclusion

```
// Fichier TestWoECN.java
public static void main(String[] args)
{
    // Creation du joueur
    Personnage leJoueur = new
        Personnage(...);
    // et des PnJ
    Personnage[] pnj = new Personnage[2];
    pnj[0] = new Guerrier(...);
    pnj[1] = new Voleur(...);
    Magicien[] lesMages[2];
    lesMages[0] = new Magicien(...);
```

## Méthodes appelées ?

```
// Fichier TestWoECN.java
// ...
leJoueur.reconstrer();
pnj[0].reconstrer();
pnj[1].reconstrer();
lesMages[0].reconstrer();
lesMages[1].reconstrer();
// ...
```

# Polymorphisme

Nous venons de détailler :

- l'héritage de type dans une hiérarchie de classes, et la notion associée de surclassement
- la notion de résolution dynamique des liens

## Polymorphisme

Ces deux notions permettent de mettre en œuvre le **polymorphisme** (nous y reviendrons un peu plus tard) :

- il permet de manipuler des objets d'une classe comme si ils étaient d'une autre classe
- et donc la "même" méthode peut s'exécuter de façon différente selon la donnée à laquelle il s'applique (le choix a lieu lors de l'exécution)

C'est ce que nous venons de voir avec les différents comportements de la méthode **rencontrer** suivant les **types effectifs** des variables

# Héritage et constructeurs

## Rappel sur les constructeurs

Le rôle d'un constructeur est d'**initialiser les attributs**, ce qui revient à :

- **donner une valeur aux attributs de type simple**, p. ex.

```
// Fichier Personnage.java dans le constructeur ...  
ptVie = 100;
```

- **appeler le constructeur des attributs de type évolué ou de type objet**, p. ex.

```
// Dans le constructeur d'une classe ayant  
// un attribut de type Date (objet) ...  
maDate = new Date(2014, 9, 15); // 15 Septembre 2014
```

## Rappel sur l'héritage

Grâce au mécanisme de l'héritage, les sous-classes **récupèrent tous les membres** (attributs et méthodes) définis dans la super-classe.

Intéressons-nous aux conséquences de l'héritage et des notions associées  
(masquage, redéfinition de méthodes, etc.) sur les constructeurs

## Héritage et constructeurs II

Lors de l'instanciation d'une sous-classe, nous devons initialiser :

- les attributs **propres à la sous-classe**
- les attributs **hérités de la (ou des) super-classe(s)**

MAIS

il **N'EST PAS** à la charge du concepteur-programmeur des sous-classes de réaliser lui même l'initialisation des attributs hérités  
⇒ cela pourrait lui être interdit (notamment si les attributs sont déclarés en **private**)

L'initialisation des attributs hérités doit se faire au niveau des classes (super-classe, super-super-classe, etc.) où ils ont été explicitement définis ! ⇒ comment faire ?

## Héritage et constructeurs II

Lors de l'instanciation d'une sous-classe, nous devons initialiser :

- les attributs **propres à la sous-classe**
- les attributs **hérités de la (ou des) super-classe(s)**

### MAIS

il **N'EST PAS** à la charge du concepteur-programmeur des sous-classes de réaliser lui même l'initialisation des attributs hérités  
⇒ cela pourrait lui être interdit (notamment si les attributs sont déclarés en **private**)

L'initialisation des attributs hérités doit se faire au niveau des classes (super-classe, super-super-classe, etc.) où ils ont été explicitement définis ! ⇒ comment faire ?

### Solution

L'initialisation des attributs hérités doit donc se faire en **appelant les constructeurs des super-classes** (ces appels doivent donc être

# Appel explicite d'un constructeur de la super-classe : syntaxe

L'invocation du constructeur d'une super-classe **DOIT se faire lors de la première instruction** du constructeur de la sous-classe, au moyen du mot clé **super**.

## Syntaxe

```
public SousClasse(listeParametres)
{
    /* arguments : liste d'arguments attendus par
     * un des constructeurs de la super-classe de SousClasse
     */
    super(arguments);    !!! important, seraient super(); sans paramètres
    // initialisation des attributs propres à SousClasse
}
```

Lorsque la super-classe possède un constructeur par défaut, l'invocation explicite de ce constructeur dans la sous-classe n'est

# Appel explicite d'un constructeur de la super-classe II

## Super-classe sans constructeur par défaut

Si la super-classe **ne possède pas de constructeur par défaut** :  
**l'appel explicite** d'un de ses constructeurs est **OBLIGATOIRE**  
 dans les constructeurs de la sous-classe ⇒ la sous-classe doit donc admettre **au moins un constructeur explicite**

### Exemple : classe Personnage

```
// Personnage.java
public class Personnage {
    private int energie;
    private int ptVie;
    public Personnage(int e, int
        pv) {
        energie = e;
        ptVie = pv;
```

### Exemple : classe Guerrier

```
// Guerrier.java
public class Guerrier extends
    Personnage {
    private Arme arme;
    public Guerrier(int e,int pv,Arme
        a) {
        super(e,pv);
        arme = new Arme(a);
    }
```

# Appel explicite d'un constructeur de la super-classe III

## Super-classe avec constructeur par défaut

Si la super-classe possède un constructeur par défaut : l'appel explicite d'un de ses constructeurs est **OPTIONNEL** dans les constructeurs de la sous-classe ⇒ Java l'appellera pour vous par défaut !

### Exemple : classe Personnage

```
// Personnage.java
public class Personnage {
    private int energie;
    private int ptVie;
    public Personnage() {
        energie = 100;
        ptVie = 100;
    }
}
```

### Exemple : classe Guerrier

```
// Guerrier.java
public class Guerrier extends Personnage {
    private Arme arme;
    public Guerrier(Arme a) {
        arme = new Arme(a);
    }
}
// ...
```

# Héritage et constructeurs : bilan

## Bilan sur les constructeurs et l'héritage

- ① Chaque constructeur d'une sous-classe doit faire un appel au constructeur de sa super-classe : `super(...)`
- ② Les arguments fournis à `super(...)` doivent correspondre à un des constructeurs de la super-classe
- ③ L'appel à `super(...)` **doit être la toute première instruction** du constructeur de la sous-classe
- ④ Si l'appel à `super(...)` **n'est pas la première instruction** ou est fait plusieurs fois  $\Rightarrow$  erreur de compilation !
- ⑤ Il est **interdit de faire appel à `super(...)` en dehors** du constructeur de la sous-classe

## Héritage et constructeurs : bilan II

Et si on oublie l'appel à `super(...)` ?

- Java fait un appel automatique à `super()`
- Attention car oubli possible et certains attributs peuvent ne pas être bien initialisés (en particulier les types évolués et les objets)
- Erreur si le **constructeur par défaut** n'existe pas

Rappel sur le constructeur par défaut

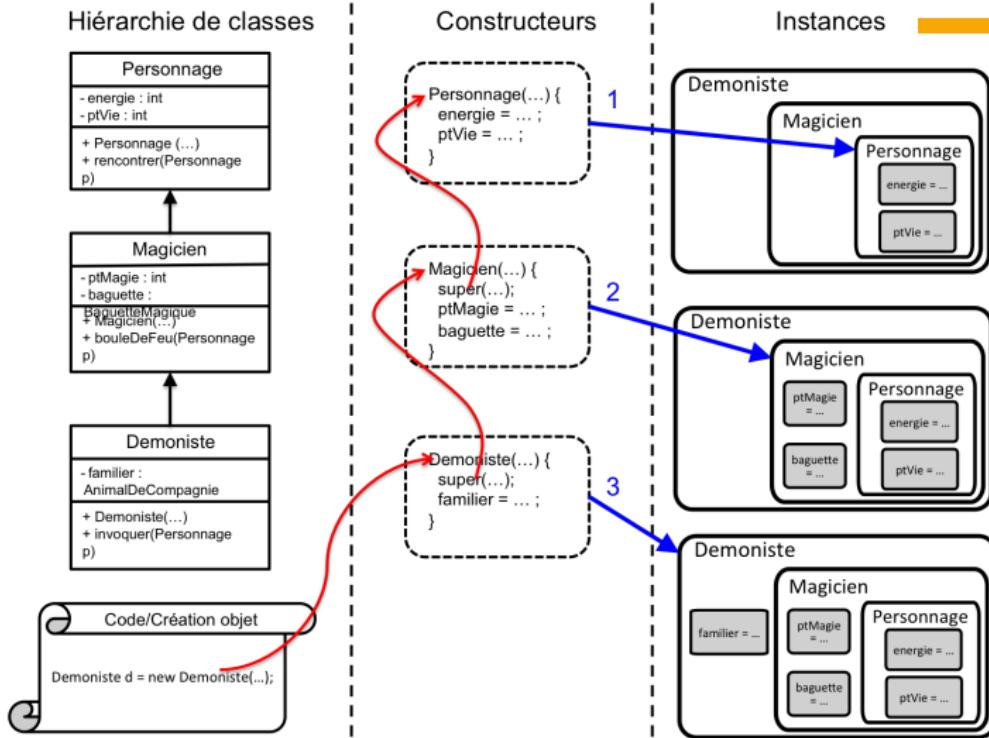
- c'est un constructeur particulier
- créé automatiquement pour chaque classe qui **n'a aucun constructeur déclaré**
- **il disparait dès qu'un autre constructeur est déclaré** par le programmeur

Recommandations

Pour éviter les problèmes dans les hiérarchies de classes, il faut :

- **toujours déclarer au moins un constructeur**
- **toujours faire l'appel à `super(...)` dans vos constructeurs !**

# Ordre d'appel des constructeurs



# Mini Quiz



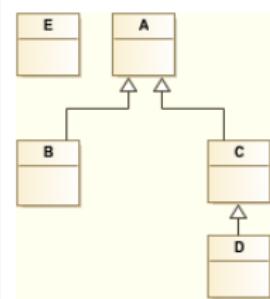
Votons à mains levées !

Quelle(s) affirmation(s)  
est/sont correcte(s) ?

- ➊ Le constructeur d'une sous-classe doit initialiser les attributs hérités de la super-classe.
  - ➋ Il est obligatoire de faire appel au constructeur de la super-classe dans le constructeur de la sous-classe.
- 2. Non, java va faire un pour vous**
- ➌ Java met en œuvre la résolution dynamique des liens.
  - ➍ Le polymorphisme est la particularité de Java qui lui permet d'être multi-plateforme.

Soit la hiérarchie de classes suivante et `a` instance de A, `b` de B, etc. Quelle(s) affirmation(s) est/sont correcte(s) ?

- ➊ `a instanceof A`
- ➋ `d instanceof D`
- ➌ `d instanceof C`
- ➍ `d = c`
- ➎ `a instanceof B`
- ➏ `b instanceof A`
- ➐ `b instanceof E`



# Compléments sur le polymorphisme

## Rappel sur le polymorphisme

En POO, le **polymorphisme** correspond au fait que :

- une instance d'une sous-classe est "substituable" à une instance d'une classe de son ascendance dans la hiérarchie des classes :
  - ▶ en argument d'une méthode
  - ▶ lors d'une affectation
  - ▶ etc.
- cette instance substituée garde néanmoins ses propriétés propres (en particulier ses méthodes)
- le choix des méthodes à invoquer se fait **lors de l'exécution du programme**, en fonction du **type réel** des instances concernées

Le polymorphisme est mis en œuvre grâce aux notions suivantes :

- l'héritage
- la résolution dynamique des liens

# Constructeurs et polymorphisme

Un constructeur est **une méthode spécifique à la construction et à l'initialisation des attributs de l'instance courante d'une classe** ⇒ un constructeur n'a pas vocation à avoir un comportement polymorphique

Mais

Il est possible d'appeler une méthode polymorphe dans le corps d'un constructeur

- c'est **déconseillé** : la méthode agit sur un objet pas forcément complètement initialisé (voir exemple à suivre)

# Constructeurs et polymorphisme II



## Problème ?

```
// Fichier A.java
public class A {
    public void m(){
        System.out.println("A");
    }
    public A(){
        m();
    }
}

// Fichier B.java
public class B extends A {
    private int b;
    public B() {super();}
    b = 1;
}
    public void m() {
        System.out.println("b vaut
        "+b);
    }
}
```

## Problème ?

```
// Fichier ExPolCons.java
public class ExPolCons {
    public static void
        main(String[] args) {
            B bb = new B();
        }
}
```

sans super(); output: b vaut 0  
super(); ajouté output : b=1

# Constructeurs et polymorphisme II



## Problème ?

```
// Fichier A.java
public class A {
    public void m(){
        System.out.println("A");
    }
    public A(){
        m();
    }
}

// Fichier B.java
public class B extends A {
    private int b;
    public B() {
        b = 1;
    }
    public void m() {
        System.out.println("b vaut
"+b);
    }
}
```

## Problème ?

```
// Fichier ExPolCons.java
public class ExPolCons {
    public static void
        main(String[] args) {
            B bb = new B();
        }
}
```

## Affichage

b vaut 0

# Retour sur des notions floues du Chapitre I

Lors du premier cours, nous avons abordé les notions d'**affichage** et de **comparaisons d'objets** via l'utilisation des méthodes `toString()` et `equals(...)` :

- `toString()` permet d'obtenir une représentation de l'objet sous forme de chaîne de caractères
- `equals(...)` permet de comparer deux objets de même type pour dire si ils sont égaux ou non

MAIS nous n'avions jamais vu d'où venaient ces méthodes ?

# Rappel sur l'affichage d'objets : la méthode `toString`

## Exemple d'utilisation de la méthode `toString`

```
// fichier PersonnageEx.java
public class PersonnageEx {
    // voir déclaration précédente de la classe PersonnageEx
    public String toString()
    {
        String res = "Personnage avec "+ptVie+" points de vie, et
                    "+energie+" points d'énergie";
        return res;
    }
}
// fichier TestPersEx.java
public class TestPersEx {
    public static void main(String[] args) {
        PersonnageEx p = new PersonnageEx(25,75)
        System.out.println(p);
    }
}
```

# Rappel sur l'affichage d'objets : la méthode `toString`

## Exemple d'utilisation de la méthode `toString`

```
// fichier PersonnageEx.java
public class PersonnageEx {
    // voir déclaration précédente de la classe PersonnageEx
    public String toString()
    {
        String res = "Personnage avec "+ptVie+" points de vie, et
                    "+energie+" points d'énergie";
        return res;
    }
}
// fichier TestPersEx.java
public class TestPersEx {
    public static void main(String[] args) {
        PersonnageEx p = new PersonnageEx(25,75)
        System.out.println(p);
    }
}
```

# Rappel comparaison d'objets : la méthode equals(...)

Code pour la classe PersonnageEx

```
// fichier PersonnageEx.java ...
public boolean equals(PersonnageEx p) {
    if(p == null) {return false;}
    else {return ((energie == p.getEnergie()) && (ptVie ==
        p.getPointsVie()));}
}
// ... dans une méthode main
PersonnageEx p1 = new PersonnageEx(150,99);
PersonnageEx p2 = new PersonnageEx(150,99);
```

Version ==

```
if(p1 == p2) {
    System.out.println("Identiques");
}
else {
    System.out.println("Différents");
}
```

Version equals

```
if(p1.equals(p2)) {
    System.out.println("Identiques");
}
else {
    System.out.println("Différents");
}
```

# Rappel comparaison d'objets : la méthode equals(...)

## Code pour la classe PersonnageEx

```
// fichier PersonnageEx.java ...
public boolean equals(PersonnageEx p) {
    if(p == null) {return false;}
    else {return ((energie == p.getEnergie()) && (ptVie ==
        p.getPointsVie()));}
}
// ... dans une methode main
PersonnageEx p1 = new PersonnageEx(150,99);
PersonnageEx p2 = new PersonnageEx(150,99);
```

### Version ==

```
if(p1 == p2) {
    System.out.println("Identiques");
}
else {
    System.out.println("Differents");
}
```

### Version equals

```
if(p1.equals(p2)) {
    System.out.println("Identiques");
}
else {
    System.out.println("Differents");
}
```

## La super-classe Object

En fait, les méthodes `toString()` et `equals(...)` sont des méthodes redéfinies !

Comment est-ce possible ?

La classe `PersonnageEx` n'hérite pourtant de `personne` :

```
// fichier PersonnageEx.java
public class PersonnageEx {  
}
```

Il n'y a pas de `extends` !!

## La super-classe Object

En fait, les méthodes `toString()` et `equals(...)` sont des méthodes redéfinies !

Comment est-ce possible ?

La classe `PersonnageEx` n'hérite pourtant de personne :

```
// fichier PersonnageEx.java
public class PersonnageEx {
}
```

`extends Object`

Il n'y a pas de `extends` !!

## La super-classe Object

Il existe en Java une **super-classe commune à toutes les classes**, qui constitue le **sommet de la hiérarchie des classes Java** : la classe `Object`

## La super-classe Object

En fait, les méthodes `toString()` et `equals(...)` sont des méthodes redéfinies !

Comment est-ce possible ?

La classe `PersonnageEx` n'hérite pourtant de personne :

```
// fichier PersonnageEx.java
public class PersonnageEx {
```

Il n'y a pas de `extends` !!

## La super-classe Object

Il existe en Java une **super-classe commune à toutes les classes**, qui constitue le **sommet de la hiérarchie des classes Java** : la classe `Object`

Toute classe que nous définissons, si elle n'hérite d'aucune classe explicitement va donc hériter AUTOMATIQUEMENT d'`Object`



# La super-classe `Object` : conséquences

## Conséquence directe

Il est tout à fait possible d'affecter **une instance de n'importe quelle classe** à une variable de type `Object` :

```
Object maVar = new NimporteQuelleClasse(...);
```

# La super-classe `Object` : conséquences

## Conséquence directe

Il est tout à fait possible d'affecter **une instance de n'importe quelle classe** à une variable de type `Object` :

```
Object maVar = new NimporteQuelleClasse(...);
```

## Conséquence directe II

Une méthode prenant en argument un objet de type `Object` va accepter toutes instances lors d'un appel :

```
// Dans une classe TestObject
public maMethode(Object o){...}
// dans un main
Personnage p = new Personnage(...);
TestObject o = new TestObject(...);
o.maMethode(p); // OK !
```

# La super-classe Object

La super classe `Object` définit entre autres les méthodes :

- `toString` : qui affiche par défaut l'adresse de l'objet sous forme de chaîne de caractères
- `equals` : qui fait par défaut une comparaison entre objets au moyen de `==` (et qui compare donc les références des objets)
- `clone` : qui permet de copier l'instance courante (mais qui ne fait pas de copie profonde, par exemple si une des attribut est une objet ou un type évolué, comme un tableau)

Dans la majorité des cas, ces méthodes par défaut ne sont pas satisfaisantes quand nous définissons nos propres classes ! Nous devons donc :

- redéfinir ces méthodes pour permettre un affichage, réaliser une comparaison ou une copie correcte de nos objets
- nous l'avons déjà fait dans le cours précédent pour `toString` et `equals`
- notons par exemple que la classe `String` redéfinit ces méthodes (c'est pourquoi il faut toujours comparer des chaînes de caractères avec `equals` !)

## Exemple de redéfinition de `equals` héritée de `Object`

Dans le cours précédent, nous avions déclaré la méthode `equals` pour la classe `PersonnageEx` avec l'entête suivant :

```
public boolean equals(PersonnageEx p){ ... }
```

Cependant, dans la classe `Object`, l'entête de la méthode `equals` est le suivant :

```
public boolean equals(Object o){ ... }
```

Donc, notre méthode de `equals` n'était pas une redéfinition de la méthode de `Object`, mais plutôt une surcharge !

Dans la majorité des cas, les surcharges de ces méthodes sont suffisantes, cependant, il est recommandé de toujours procéder par des redéfinitions en Java ! ⇒ en particulier à cause des Collections de Java qui utilisent uniquement l'entête de `equals` prenant un `Object` en paramètre (nous y reviendrons).

# Surcharge, redéfinition : rappels

Une redéfinition ("override") de méthode consiste à définir une méthode :

- de même nom
- de type de retour **compatible**, i.e. :
  - ▶ **même type pour les types de base** (`int`, `float`, `boolean`, etc.)
  - ▶ pour les objets si il y a une relation d'héritage entre les types rentrés
- avec **la même liste d'arguments** : même nombre et même type des arguments (et dans le même ordre)

## Redéfinition

```
public boolean equals(Object o){ ... }
```

# Surcharge, redéfinition : rappels

Une surcharge ("override") de méthode correspond quant à elle à la définition d'une méthode :

- **de même nom**

## Surcharge

```
public boolean equals(PersonnageEx p){ ... }
```

# Redéfinition recommandée de equals : exemple pour la classe PersonnageEx

## Attention !

Si l'on redéfinit `equals` pour la classe `PersonnageEx`, on doit pouvoir comparer tout objet de cette classe avec n'importe quel autre objet :  
`monPerso.equals("salut")` doit retourner `false` !

## Redéfinition recommandée de equals

```
public class PersonnageEx {  
    // ...  
    public boolean equals(Object o) {  
        if ( (o == null) || (o.getClass() != this.getClass()) ) {  
            // nous reviendrons plus tard sur la methode getClass()  
            return false;  
        } else {  
            PersonnageEx p = (PersonnageEx)o;  
            return ( (energie == p.getEnergie()) && (ptVie ==  
                p.getPointsVie()) );  
        }  
    }  
}
```

# Polymorphisme et le modificateur final

Ce modificateur (ou mot clé) permet d'indiquer au compilateur Java les éléments des classes qui ne doivent pas être **redéfinis/modifiés/étendus** !

Le modificateur **final** :

- peut s'appliquer à une classe, une méthode, un attribut, une variable
- surtout utilisé pour les variables
- moins courant pour les classes et les méthodes

# Classes finales

Lorsque l'on ajoute `final` à la déclaration d'une classe : **il devient interdit d'hériter de cette classe !**

## Hiérarchie des classes Personnages

On veut interdire que la classe `Magicien` puisse avoir des sous-classes :

```
public final class Magicien {  
    // ...  
}
```

Il devient ainsi impossible de déclarer notre classe `Demoniste` !!

## Erreur de compilation !

```
public class Demoniste extends Magicien { // ERREUR !!  
    // ...  
}
```

# Méthodes finales

Lorsque l'on ajoute **final** à la déclaration d'une méthode : **il devient interdit de la redéfinir dans une sous-classe !**

## Hiérarchie des classes Personnages

On aimerait avoir une seule méthode retirant un point d'énergie à nos personnages : celle de la classe **Personnage** :

```
public class Personnage {  
    // ...  
    public final void actionSimple() {  
        energie--;  
    }  
}
```

Si l'on tente de redéfinir la méthode **actionSimple()** dans une sous-classe de **Personnage** (n'importe laquelle) ⇒ **erreur du compilateur !**

# Classes et Méthodes finales

L'utilisation du modificateur **final** devant des classes ou des méthodes est restrictif :

- cela empêche l'amélioration de classes ou méthodes par redéfinition
- mais cela permet de fixer une fois pour toute le comportement d'une classe ou d'une méthode

Les chaînes de caractères sont finales !

En Java, la classe **String** est déclarée comme finale, il est donc impossible d'hériter de cette classe afin de lui donner un comportement particulier qui nous arrangerait :

```
public class MaString extends String{  
    // erreur de compilation !!  
}
```

Mais ainsi on assure que tous les programmeurs auront le même comportement lorsqu'ils utilisent des objets de type **String**

## Variables finales

Si l'on déclare un attribut, une variable ou un paramètre de méthode comme `final` : **il devient interdit de lui affecter une valeur plus d'une fois !**

Un attribut `final` peut donc être initialisé dans le constructeur, mais ne peut plus être modifié par la suite (sinon erreur de compilation) !

Attention toutefois !!

Rendre une variable `final` empêche de lui affecter une nouvelle valeur, mais **pas de modifier l'objet référencé par cette variable !**

# Exemple de variable finale et modification d'objet référencé

Voir classe [TestModifFinal](#) 

Exemple de problème potentiel avec les variables finales

```
public class Personnage {  
    // voir classe definie avec attributs et accessseurs/modificateurs  
}  
  
public class TestModifFinal {  
    public static void modifier(final Personnage pers) {  
        pers.setEnergie(150); // OK mais modifie l'objet 'pers' passe en  
        // parametre meme si il est declare en final !!  
  
        pers = new Personnage(...); // Interdit !! On essaye de modifier la  
        // variable 'pers' declaree comme final --> erreur de compilation  
    }  
  
    public static void main(String[] args) {  
        Personnage p = new Personnage(50,100);  
        modifier(p);  
    }  
}
```

## Le modificateur static

Après avoir parlé du modificateur **final**, intéressons nous au dernier mot clé que nous avons utilisé sans l'expliquer : le modificateur **static**

En Java, le mot clé **static** peut être ajouté devant :

- des attributs de classe (pas devant des variables locales, ni devant des paramètres de méthodes) ⇒ dans ce cas on parle de **variables statiques**, **attributus statiques** ou **membres statiques**
- des méthodes (en particulier la méthode **main**)
- un bloc de code (rare, hors programme)
- très très rarement devant une classe (en fait **uniquement** pour des classes internes : classes définies à l'intérieur d'une autre classe, **hors programme**)

## Le modificateur static II

### static et variables

Si l'on ajoute **static** devant une variable :

- la valeur de la variable est **partagée entre toutes les instances de la classe**
- interdit pour les variables locales (et pour les paramètres)

### static et méthodes

Si l'on ajoute **static** devant une méthode :

- il est possible d'appeler la méthode (appelée **méthode statique**) sans créer d'objet ! On l'appelle alors directement en utilisant le nom de la classe :

### Appel méthode statique

`NomClasse.nomMéthodeStatique(listeArguments)`

# Membres statiques = Variables de classe

Jusqu'ici nous avons parlé de différents types de variables :

- ❶ **Variables d'instance (= attributs)** : représentent les caractéristiques propres d'un objet

Variable d'instance = attribut

```
public class Personnage {  
    // represente le niveau d'energie du personnage  
    private int energie;  
}
```

# Membres statiques = Variables de classe

Jusqu'ici nous avons parlé de différents types de variables :

- ① **Variables d'instance (= attributs)** : représentent les caractéristiques propres d'un objet
- ② **Variables locales** : déclarées à l'intérieur d'une méthode

## Variable locale

```
public class Test {  
    public static void main(String[] args) {  
        float f = 5.23; // f est une variable locale  
        System.out.println(f*2);  
    }  
}
```

# Membres statiques = Variables de classe

Jusqu'ici nous avons parlé de différents types de variables :

- ❶ **Variables d'instance** (= attributs) : représentent les caractéristiques propres d'un objet
- ❷ **Variables locales** : déclarées à l'intérieur d'une méthode
- ❸ **Paramètres** : permettent d'envoyer des données à l'intérieur d'une méthode, s'utilisent comme des variables locales

## Paramètre

```
public class Personnage {  
    // ...  
    public void setEnergie(int e) {  
        e = 2*e; // e est un parametre de la methode  
        energie = e; // equivalent a this.energie = e;  
    }  
}
```

# Membres statiques = Variables de classe

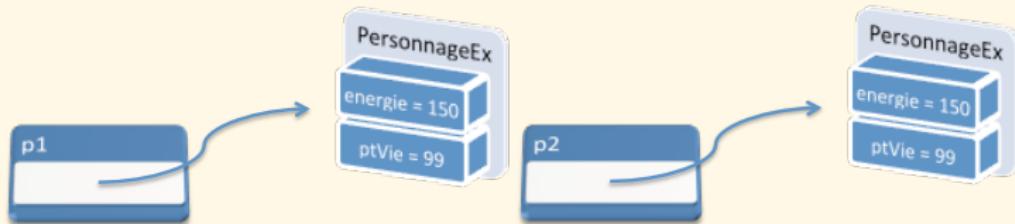
Jusqu'ici nous avons parlé de différents types de variables :

- ① **Variables d'instance** (= attributs) : représentent les caractéristiques propres d'un objet
- ② **Variables locales** : déclarées (et visibles uniquement) à l'intérieur d'une méthode
- ③ **Paramètres** : permettent d'envoyer des données à l'intérieur d'une méthode, s'utilisent comme des variables locales
- ④ **Nouveau : Variables statiques ou Variables de classe** : indiquées par le modificateur **static** :
  - ▶ ressemblent aux attributs (variables d'instance)
  - ▶ sont déclarées en dehors des méthodes (comme les attributs)
  - ▶ sont visibles partout dans la classe
  - ▶ sont héritées par les sous-classes

# Variable d'instance (attribut) vs. variable de classe

## Variables d'instance (attributs)

- représentent les caractéristiques propres d'un objet
- ont chacune une zone mémoire réservée lors de la création de l'objet (avec l'appel à `new`)
- conséquence : **chaque objet possède un espace mémoire qui lui est propre pour chacun de ses attributs !**



# Variable d'instance (attribut) vs. variable de classe

## Variables de classe (variables statiques)

- sont précédées de **static** lors de leurs déclarations
- ont un espace mémoire réservé lors du chargement de la classe par Java
- aucun espace mémoire supplémentaire réservé lors de la création d'instances de la classe avec **new**
- conséquence : **toutes les instances de la classe se réfèrent à la même zone mémoire lorsqu'ils accèdent à la variable de classe**

# Exemple de variable de classe



## Exemple de problème potentiel avec les variables statiques

```
public class A {  
    // variable d'instance (= attribut)  
    private int monAttrib;  
    // variable de classe /!\ initialisation en dehors d'une méthode  
    static public int maVarClasse = 100;  -> Valeur en dehors de constructor  
    // constructeur  
    public A(int a) {  
        monAttrib = a;  
    }  
    // méthodes  
    public void incrVariables() {  
        monAttrib++; // incremente l'attribut  
        maVarClasse++; // increment la variable de classe  
    }  
    public void affiche() {  
        System.out.println("monAttrib: "+monAttrib+"-maVarClasse:  
        "+maVarClasse);  
    }  
}
```

# Exemple de variable de classe II



## Exemple de problème potentiel avec les variables statiques

```
public class ExVarClasse {  
    public static void main(String[] args) {  
        A monObjA = new A(25); // creation d'un objet de type A  
        monObjA.incrVariables(); // appel de la methode d'increment des  
        variables  
        monObjA.affiche(); // affichage de monObjA  
  
        A.maVarClasse = 0; // modification de la variable de classe !  
  
        A monObjB = new A(1); // creation d'un objet de type A  
        monObjB.incrVariables();  
        monObjB.affiche();  
  
        // exemple  
        monObjA.incrVariables();  
        monObjA.affiche();  
        monObjB.affiche();  
    }  
}
```

# Exemple de variable de classe III



## Résultat

```
monAttrib: 26 - maVarClasse: 101
monAttrib: 2 - maVarClasse: 1
monAttrib: 27 - maVarClasse: 2
monAttrib: 2 - maVarClasse: 2
```

# Pourquoi utiliser static ?

Modification d'une variable d'instance (attribut)

La valeur de l'attribut est modifiée **uniquement pour l'objet courant** !

Modification d'une variable de classe (statique)

La valeur est modifiée **pour toutes les instances de la classe** !

À quoi sert une variable statique ? Quand doit-on les utiliser ?

- ① Bonne raison d'utiliser une variable statique : représentation d'une valeur commune pour tous les objets de la classe
- ② Mauvaise raison d'utiliser une variable statique : programmer de manière non-orientée objet en Java

## Exemple : valeur commune à toutes les instances d'une classe

Exercice : écrire une classe `Employe` avec un âge de départ à la retraite de 65 ans

Considérons les deux cas suivants :

- avec une variable d'instance (attribut) `ageRetraite`
- avec une variable de classe `ageRetraite`

# Classe EmployeVI



## Exemple de classe avec variable d'instance

```
public class EmployeVI {  
    private String nom;  
    private int ageRetraite;  
  
    public EmployeVI(String n, int ar) {  
        this.nom = n;  
        this.ageRetraite = ar;  
    }  
  
    public void setAgeRetraite(int nar) {  
        this.ageRetraite = nar;  
    }  
  
    public void afficheEmploye() {  
        System.out.println("L'employe "+nom+" part a la retraite a:  
        "+ageRetraite);  
    }  
}
```

# Utilisation de EmployeVI



## Utilisation de la classe EmployeVI

```
public class EntrepriseVI {

    public static void main(String[] args) {
        EmployeVI[] employes = new EmployeVI[5];
        employes[0] = new EmployeVI("John Doe", 65);
        employes[1] = new EmployeVI("Mary Lee", 65);
        employes[2] = new EmployeVI("Tim Johnson", 65);
        employes[3] = new EmployeVI("Elizabeth Steed", 65);
        employes[4] = new EmployeVI("Ashley Martin", 65);

        // Si un jour l'age de la retraite est modifie
        // il va falloir modifier la valeur de l'attribut
        // pour TOUS les objets !
        for(int i= 0;i<employes.length; i++) {
            employes[i].setAgeRetraite(67);
        }
    }
}
```

# Classe EmployeVC



## Exemple de classe avec variable de classe

```
public class EmployeVC {  
    private String nom;  
    public static int ageRetraite = 65; // on doit le mettre en public  
    dans cet exemple !  
  
    public EmployeVC(String n) {  
        this.nom = n;  
    }  
  
    public void afficheEmploye() {  
        System.out.println("L'employé "+nom+" part à la retraite à:  
        "+ageRetraite);  
    }  
}
```

# Utilisation de EmployeVC



## Utilisation de la classe EmployeVC

```
public class EntrepriseVC {  
  
    public static void main(String[] args) {  
        EmployeVC[] employes = new EmployeVC[5];  
        employes[0] = new EmployeVC("John Doe");  
        employes[1] = new EmployeVC("Mary Lee");  
        employes[2] = new EmployeVC("Tim Johnson");  
        employes[3] = new EmployeVC("Elizabeth Steed");  
        employes[4] = new EmployeVC("Ashley Martin");  
  
        // Si un jour l'age de la retraite est modifie  
        // il suffit de la modifier une seule fois pour tous les objets!  
        EmployeVC.ageRetraite = 67;  
        // ou bien  
        employes[0].ageRetraite = 67; // c'est equivalent !  
    }  
}
```

## Déclaration de constantes : final et static

Déclaration de constantes communes à toutes les instances d'une classe :

- inutile de stocker une valeur pour chaque instance
- la valeur ne sera pas modifiée une fois définie
- **les déclarer comme final static**

Exemple de constante pour toutes les instances d'une classe

```
public class Planete {  
    // G = constante gravitationnelle  
  
    // Une variable finale (constante) pour chaque planete  
    private final double G = 6.674E-8;  
  
    // Une variable finale (constante) pour toutes les planetes !  
    // c'est mieux !  
    private final static double G = 6.674E-8;  
}
```

# Explication d'un mystère Java

Nous sommes maintenant en mesure d'expliquer pourquoi certaines méthodes que nous utilisons ont une syntaxe étrange : p. ex.

`System.out.println`

## Analysons `System.out.println`

- `System` : classe prédéfinie Java (on sait que c'est une classe car son nom débute par une majuscule)
- `out` :
  - ▶ variable statique de la classe `System`
  - ▶ c'est un objet car il est suivi d'un point '.'
- `println` : méthode de l'objet `out` qui prend une chaîne de caractère (classe `String`) en paramètre

# Méthodes statiques

Comme pour les variables, si l'on ajoute **static** devant une méthode, alors **on peut l'appeler sans avoir à passer par un objet !**



## Exemple de méthode static

```
public class A {
    public static void
        methodeStatique()
    {
        System.out.println("Methode
                           statique");
    }

    public void methodeClassique()
    {
        System.out.println("Methode
                           classique");
    }
}
```

Juste pour «static» →

## Exemple d'appel de méthode static

```
public class ExMethodeStat {
    public static void main(String[]
                           args) {
        A.methodeStatique(); // OK
        A.methodeClassique(); // Erreur de compilation
    }
}
```

```
A monObjA = new A();
monObjA.methodeStatique(); // OK (alternative non recommandée)
monObjA.methodeClassique();
// OK (comme d'habitude)
```

## Méthodes statiques : contraintes

Nous venons de voir qu'une méthode statique peut être appelée directement par la classe et sans passer par un objet, ainsi :

- il est possible que l'objet `this` n'existe pas !
- on ne peut donc permettre l'accès aux attributs dans le corps d'une méthode statique !
- si l'objet `this` n'existe pas  $\Rightarrow$  ses attributs n'ont pas été créés

Conséquence sur les méthodes statiques :

Au sein d'une classe, une méthode statique peut uniquement accéder à :

- d'autres méthodes statiques !
- des variables statiques !

### Attention !

Il est possible d'appeler une méthode statique en utilisant un objet  
**MAIS** c'est fortement déconseillé  $\Rightarrow$  dans ce cas il ne saute pas aux veux que la méthode est statique !

# Illustration des contraintes pour des méthodes statiques

## Erreurs potentielles liées à une méthode statique

```
public class A {  
    private int monAttrib;  
    private static int maVarStatique;  
  
    public void methodeClassique() {  
        System.out.println(monAttrib);          // OK  
        System.out.println(maVarStatique);       // OK  
        methodeStatique();                     // OK  
    }  
  
    public static void methodeStatique() {  
        System.out.println(monAttrib);          // Erreur !  
        System.out.println(maVarStatique);       // OK  
        methodeClassique();                   // Erreur !  
        methodeStatique();                    // Autorise mais idiot  
  
        A mA = new A();  
        mA.methodeClassique();                // OK  
    }  
}
```

# Intérêt des méthodes statiques ?

- méthode statique : méthode non liée à un objet
- intérêt : méthode “utilitaire”
- exemple : la classe `Math` de Java qui contient tout un ensemble d’opérations mathématiques basiques (racine carrée, valeur absolue, sinus, etc.)
- `JavaDoc en ligne :`  
`http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html`
- autre exemple : la méthode `main(String[] args)`

# Utilisation des méthodes et variables statiques

Évitez la prolifération de **static** !

On utilise le mot clé **static** uniquement dans des situations particulières :

- déclaration d'une constante : attribut **final static** (situation très courante)
- déclaration d'un attribut commun à toute une classe : attribut **static** (plus rare)
- déclaration d'une méthode utilitaire qu'il est inutile (voire illogique) de lier à un objet : méthode **static invocable sans objet** (également plus rare)

## Méthodes auxiliaires de la méthode main

Nous savions déjà que la méthode `main` était déclarée comme `static` et nous savons maintenant ce que cela signifie.

Il est donc logique que les méthodes auxiliaires de la méthode `main` soient elles aussi déclarées comme étant `static` !

L'entête de la méthode `main` est forcément le suivant :

`public static void main(String[] args)`

Puisque `main` est forcément statique, cela implique :

- elle ne peut accéder à l'objet `this`
- elle ne peut accéder à des méthodes/variables d'instance
- elle peut seulement accéder à des méthodes/variables statiques !

Notez toutefois qu'en dehors de cela, la classe de la méthode `main` est une classe comme les autres, elle peut avoir :

- des constructeurs
- des attributs
- des méthodes "classiques" (i.e. non statiques)

# Mini Quiz



Votons à mains levées !

Quelle(s) est/sont les affirmation(s) correcte(s) ?

- ➊ Une variable de classe est propre à une instance.
- ➋ Une méthode statique peut être invoquée directement par la classe.
- ➌ Il n'est pas possible d'avoir des constantes en Java. **`final static`**
- ➍ Les méthodes statiques peuvent accéder aux attributs d'une instance.
- ➎ Il est possible d'appeler une méthode non statique depuis la méthode **`main`**  
**non car «main» est static aussi**

Quelle(s) est/sont les affirmation(s) correcte(s) ?

- ➊ Il est interdit de déclarer des classes finales.
- ➋ On ne peut pas redéfinir une méthode statique.
- ➌ `public boolean equals(Personnage p)` est une redéfinition de la méthode `equals` de la classe `Object`.  
surcharge car equals de paramètre Object o
- ➍ Il est conseillé de redéfinir `toString` dans toutes vos classes.
- ➎ L'instruction `if(a = b)` est une comparaison. ==