

## Introduction à la Programmation : Héritage

Laboratoire d'Intelligence Artificielle  
Faculté I&C

## Pendant l'heure de cours

- ▶ Petit rappel des points importants
- ▶ Fiches résumé : [héritage](#)
- ▶ Approfondissements :
  - ▶ Etude de cas (Être ou avoir ?)
  - ▶ [Droit d'accès protégé](#) (17)
  - ▶ [Complément sur les paquetages](#)(51)

## Objectifs du cours de cette semaine

- ▶ Introduire la notion d'**héritage** en POO
- ▶ Comment cette notion se met en pratique en Java
- ▶ Héritage et droits d'accès
- ▶ Constructeurs et héritage
- ▶ Masquage dans une hiérarchie de classes
- ▶ Introduction au polymorphisme (résolution dynamique des liens)

## Passons aux choses sérieuses ...

Vous maîtrisez maintenant les calculs de scores pour votre jeu en ligne 🎮 Il est temps de le peupler un peu



# Ebauche de conception....

- ▶ Une classe pour le programme principal : **Jeu**
- ▶ Une classe pour représenter le **Joueur**
- ▶ Une classe pour représenter une **Partie** (avec le joueur et tous les personnages/avatars virtuels qu'il va rencontrer)
- ▶ Quatre classes pour des types d'avatars particuliers :
  1. des **Orcs**,
  2. des **Elfes**,
  3. des **Magiciens** et,
  4. des **Sorciers**.
- ▶ et plein de petites classes utilitaires pour représenter des accessoires : **Arme**, **Baguette** etc.

# Classes pour les Avatars

class Orc

```
String nom
int energie
int dureeVie
Arme arme
rencontrer(Joueur)
```

class Elfe

```
String nom
int energie
int dureeVie
rencontrer(Joueur)
```

class Magicien

```
String nom
int energie
int dureeVie
Baguette baguette
rencontrer(Joueur)
```

class Sorcier

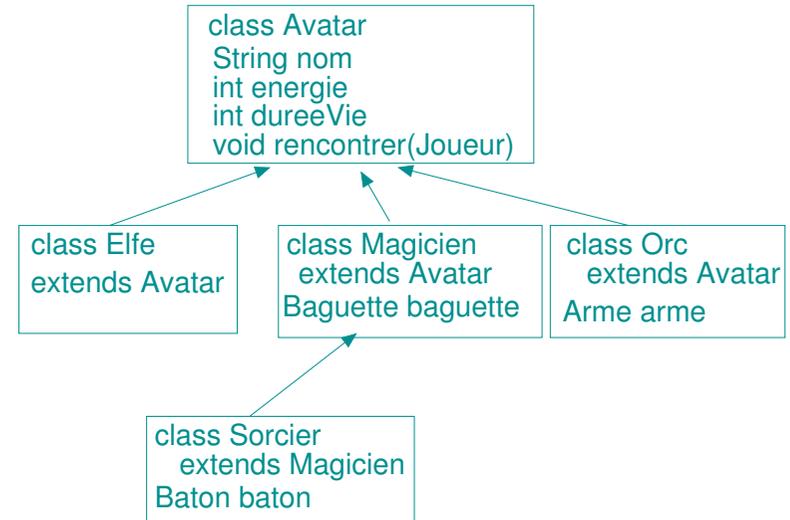
```
String nom
int energie
int dureeVie
Baguette baguette
Baton baton
rencontrer(Joueur)
```

# Classes pour les Avatars



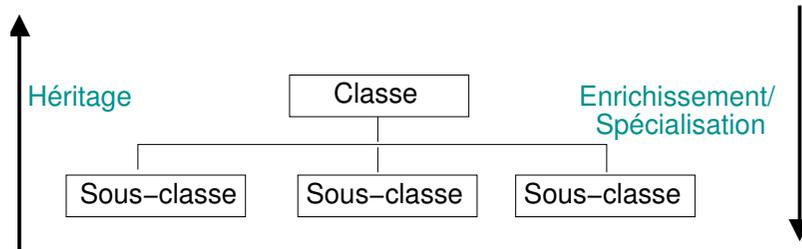
On duplique beaucoup de code d'une classe à l'autre (espace et temps perdus ... mais surtout, problèmes de maintenance) !

# Héritage



## Héritage (1)

- ▶ Après les notions d'**encapsulation** et d'**abstraction**, le troisième aspect essentiel des objets est la notion d'**héritage**
- ▶ L'héritage est une **technique extrêmement efficace** pour créer des classes plus spécialisées, appelées **sous-classes**, à partir de classes plus générales déjà existantes, appelées **sur-classes**.



## Héritage (3)

L'héritage permet donc :

- ▶ D'**expliciter des relations** structurelles et sémantiques entre classes.
- ▶ De **réduire les redondances** de description et de stockage des propriétés.

Attention, l'héritage doit être utilisé :

- ▶ pour décrire une relation **"est-un"** ("is-a") entre les classes ;
- ▶ il ne doit **jamais** décrire une relation **"a-un"** ("possède-un"/ "has-a").

## Héritage (2)

Plus précisément, lorsqu'une sous-classe **SousClasse** est créée à partir d'une classe **SuperClasse**, **SousClasse** va **hériter** de l'ensemble :

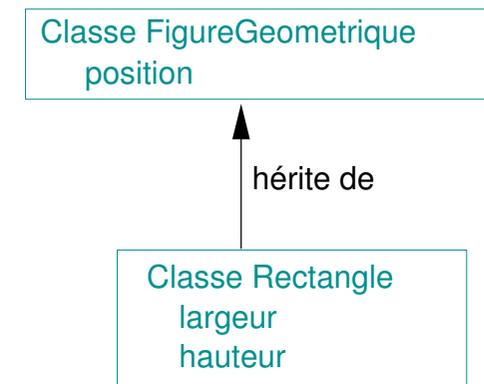
- ▶ des attributs de **SuperClasse**
  - ▶ des méthodes de **SuperClasse**
- ↳ Les attributs et méthodes de **SuperClasse**, aussi appelés **membres** de **SuperClasse**, vont être disponibles pour **SousClasse** sans que l'on ait besoin de les redéfinir explicitement dans **SousClasse**.

De plus, des attributs et/ou méthodes supplémentaires peuvent être définis par la sous-classe **SousClasse**

- ↳ Ces nouveaux membres constituent l'**enrichissement** apporté par cette sous-classe.

## Héritage (4)

Par exemple, grâce à l'héritage, on peut étendre une classe **FormeGeometrique**, caractérisée par un attribut **position**, avec une sous-classe **Rectangle** ayant pour attributs **largeur** et **hauteur**.



- ↳ Un rectangle **"est-une"** forme géométrique
- ↳ Une forme géométrique **"possède-une"** position

## Héritage (5)

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

- ▶ Les attributs et méthodes de l'ensemble des classes parentes (classe parente, classe parente de la parente etc ...)

La notion d'**enrichissement par héritage** :

- ▶ Crée un **réseau de dépendances** entre classes.
- ▶ Ce réseau est organisé en une **structure arborescente** où chacun des noeuds hérite des propriétés de l'ensemble des noeuds du chemin remontant jusqu'à la racine.

☞ Ce réseau de dépendance définit une **hiérarchie de classes**

## Super- et sous-classes

Une **super-classe** :

- ▶ est une classe "parente"
- ▶ déclare les variables/méthodes communes
- ▶ peut avoir plusieurs sous-classes

Une **sous-classe** est :

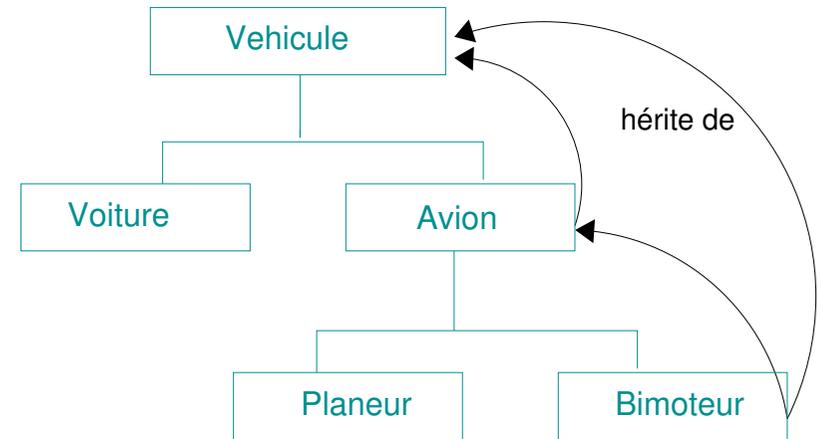
- ▶ une classe "enfant"
- ▶ étend **une seule** super-classe
- ▶ hérite des **variables**, des **méthodes** et du **type** de la super-classe

Une variable/méthode héritée peut s'utiliser :

- ▶ comme si elle était déclarée dans la sous-classe au lieu de la super-classe (en fonction des droits d'accès, voir plus loin)

☞ On évite ainsi la **duplication de code**

## Hiérarchie de classes



## Passons à la pratique...

Définition d'une sous-classe en Java :

**Syntaxe** :

```

class NomClasseEnfant extends NomClasseParente
{
    /* Déclaration des attributs et méthodes
       spécifiques à la sous-classe */
    //...
};
  
```

**Exemple** :

```

class Rectangle extends FormeGeometrique
{
    int largeur; // attributs spécifiques
    int hauteur;
    //...
};
  
```

## Accès aux membres d'une sous-classe

Jusqu'à maintenant, l'accès aux membres (attributs et méthodes) d'une classe pouvait être :

- ▶ soit **public** : visibilité totale à l'intérieur et à l'extérieur de la classe (mot-clé `public`)
- ▶ soit **privé** : visibilité uniquement à l'intérieur de la classe (mot-clé `private`)
- ▶ soit par défaut (aucun modificateur) : visibilité depuis toutes les classes du même paquetage (est aussi valable pour le paquetage par défaut que vous utilisez en TP)

Un troisième type d'accès régit l'accès aux attributs/méthodes au sein d'une hiérarchie de classes :

- ▶ l'accès **protégé** : assure la visibilité des membres d'une classe dans les classes de sa descendance (et dans autres classes du même paquetage). Le mot clé est «`protected`».

(Voir <https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html> pour une description exhaustive)

## Utilisation des droits d'accès

- ▶ Membres **publics** : accessibles pour les **programmeurs utilisateurs** de la classe
- ▶ Membres **protégés** : accessibles aux **programmeurs d'extensions** par héritage de la classe (ou travaillant dans le même paquetage)
- ▶ Membres **privés** : pour le **programmeur de la classe** : structure interne, **modifiable** si nécessaire **sans répercussions** ni sur les utilisateurs ni sur les autres programmeurs.

## Droits d'accès protégé

- ▶ Une sous-classe n'a **pas de droit d'accès** aux membres (attributs ou méthodes) **privés** hérités de ses super-classes
  - ☞ elle doit alors utiliser les getter/setters prévus dans la super-classe
- ▶ Si une super-classe veut permettre à ses sous-classes d'accéder à un membre donné, elle doit le déclarer non pas comme privé (`private`), mais comme protégé (`protected`).

**Attention** : La définition d'attributs protégés nuit à une bonne encapsulation d'autant plus qu'en Java un membre protégé est aussi accessible par toutes les classes d'un même paquetage

## Constructeurs et héritage

Lors de l'instanciation d'une sous-classe, il faut initialiser :

- ▶ les attributs **propres à la sous-classe**
- ▶ les attributs **hérités des super-classes**

**MAIS...**

...il ne doit pas être à la charge du concepteur des sous-classes de réaliser lui-même l'**initialisation des attributs hérités**

L'**accès** à ces attributs peut notamment être **interdit** ! (`private`)

L'initialisation des attributs hérités doit donc se faire au niveau des classes où ils sont explicitement définis.

**Solution** : l'initialisation des attributs hérités doit se faire en invoquant les **constructeurs des super-classes**.

## Constructeurs et héritage (2)

L'invocation du constructeur de la super-classe se fait au **début du corps du constructeur de la sous-classe** au moyen du mot clé `super`.

### Syntaxe :

```
SousClasse(liste d'arguments)
{
    super(...);
    // corps du constructeur
}
```

### Règles :

1. Chaque constructeur d'une sous-classe *doit* appeler `super(...)`
2. Les arguments fournis à `super` doivent être ceux d'au moins un des constructeur de la super-classe.
3. L'appel doit être la toute **1ère instruction**
4. Erreur si l'appel vient plus tard ou 2 fois
5. Aucune autre méthode ne peut appeler `super(...)`

## Constructeurs et héritage : exemple (2)

```
class Rectangle {
    private double largeur;
    private double hauteur;
    // il y a un constructeur par default !
    public Rectangle()
        { largeur = 0; hauteur = 0; }
    // le reste de la classe...
};

class Rectangle3D extends Rectangle {
    private double profondeur;
    public Rectangle3D(double p)
        {profondeur=p;}
    // le reste de la classe...
}
```

Ici il n'est pas nécessaire d'invoquer explicitement le constructeur de la classe parente puisque celle-ci admet un constructeur par défaut.

## Appel obligatoire à super(...) (2)



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

- ▶ Appel automatique à `super()`
- ▶ Pratique parfois, mais erreur si le **constructeur sans paramètres** n'existe pas

Rappel : le constructeur sans paramètres est particulier

- ▶ Il existe par défaut pour chaque classe qui n'a aucun autre constructeur
- ▶ Il disparaît dès qu'il y a un autre constructeur

Pour éviter des problèmes avec les hiérarchies de classes :

- ▶ Toujours déclarer au moins un constructeur
- ▶ Toujours faire l'appel à `super(...)`

## Encore un exemple

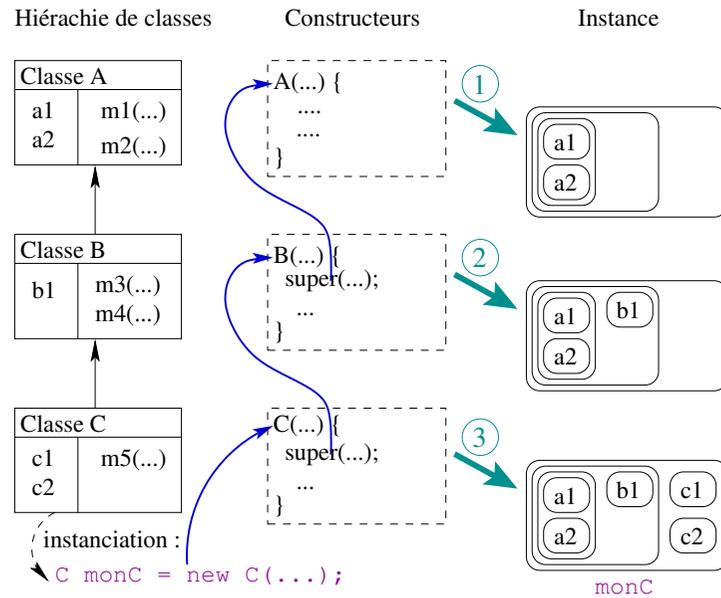
Il n'est pas nécessaire d'avoir des attributs supplémentaires...

```
class Carre extends Rectangle {

    public Carre(double taille)
    {
        super(taille, taille);
    }

    // et c'est tout ! (sauf s'il y avait des
    // "methodes set" dans Rectangle)
}
```

# Ordre d'appel des constructeurs



# super (...) ≠ new

Nous connaissons maintenant deux façons d'appeler le constructeur d'une classe.

1. `new Rectangle(uneLargeur, uneHauteur)`
  - ▶ Façon générale de construire un objet
  - ▶ Réserve de mémoire + exécution des instructions
2. `super(largeur, hauteur)`
  - ▶ invoqué uniquement par le constructeur d'une sous-classe de `Rectangle`
  - ▶ Exécution des instructions seulement
  - ▶ Réserve de mémoire déjà faite dans la sous-classe

# super (...) ≠ this

- ▶ Il existe une troisième façon d'invoquer un constructeur : `this(...)`
- ▶ s'utilise en cas de surcharge des constructeurs
- ▶ permet à un constructeur d'invoquer un autre constructeur de la même classe
- ▶ doit être la première instruction du constructeur
- ▶ ne peut pas cohabiter avec une invocation à `super(...)`

# super (...) ≠ this (2)

```
class Rectangle
{
    private double largeur;
    private double hauteur;

    public Rectangle()
    {
        largeur = 0.0;
        hauteur = 0.0;
    }

    public Rectangle(double uneLargeur, double uneHauteur)
    {
        largeur = uneLargeur;
        hauteur = uneHauteur;
    }

    public Rectangle(double uneValeur)
    {
        // invoque la constructeur precedent
        this(uneValeur, uneValeur);
    }
}
```

## Héritage et constructeur de copie

Le constructeur de copie d'une sous-classe doit invoquer explicitement le constructeur **de copie** de la super-classe

- ↳ Sinon c'est le constructeur **par défaut** de la super-classe qui est appelé !!

Exemple :

```
RectangleColore(RectangleColore autreRectangleColore)
{
    // Appel au constructeur de copie de la super-classe
    super(autreRectangleColore);
    // ....
}
```

## Appel à une méthode dans une hiérarchie

Exemple : appel à `rencontrer(Joueur)` sur un objet de type `Sorcier`

```
Joueur toto = new Joueur(...);
Sorcier oz = new Sorcier(...);

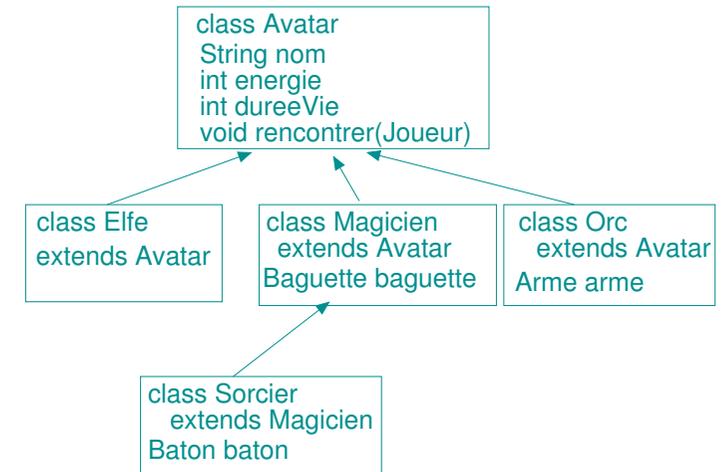
oz.rencontrer(toto);
```

1. Recherche de la méthode dans la classe de l'objet
2. Pas trouvée dans `Sorcier`
3. Recherche de la méthode dans la hiérarchie, en commençant par la super-classe directe
4. Pas trouvée dans `Magicien`
5. Trouvée dans `Avatar`, exécution de la méthode

Conclusions :

- ▶ C'est la variable/méthode de la classe **la plus proche** de l'objet qui sera utilisée
- ▶ et si un `Orc` avait une autre façon de rencontrer le joueur ?

## Appel à une méthode dans une hiérarchie



Comment se passe l'appel à la méthode `rencontrer(Joueur)` sur un objet de la sous-classe `Sorcier` ?

## Les Orc font bande à part

▶ Pour un `Orc`

```
class Orc
{
    //...
    private Arme monArme;
    public void rencontrer(Joueur lePauvre)
    {
        frapper(lePauvre); // avec monArme bien sur
    }
}
```

▶ Pour tous les autres :

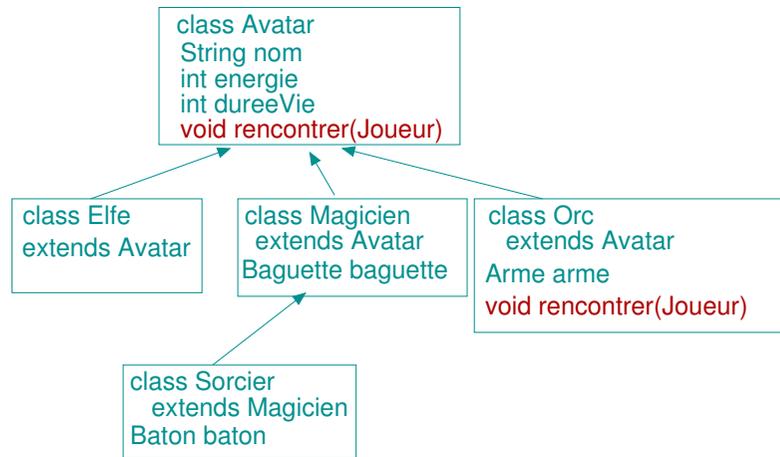
```
class Avatar
{
    //...
    public void rencontrer(Joueur unJoueur)
    {
        saluer(leJoueur);
    }
}
```



Faut-il re-concevoir toute la hiérarchie ?

- ↳ Non, on ajoute simplement une méthode `rencontrer(Joueur)` spéciale dans la sous-classe `Orc`

## Les Orc font bande à part : Masquage



## Masquage dans une hiérarchie

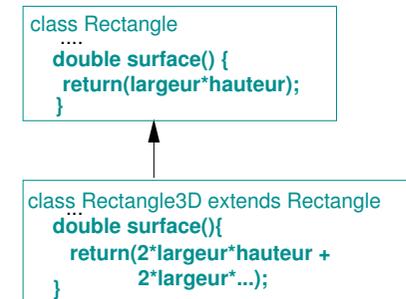
**Masquage** ("shadowing" pour les variables/ "overriding" pour les méthodes) :

- ▶ Même nom de variable utilisé sur plusieurs niveaux
- ▶ Même signature de méthode utilisée sur plusieurs niveaux
- ▶ Peu courant pour les variables (... et à éviter !)
- ▶ Très utiles pour les méthodes

## Masquage dans une hiérarchie : exemple simple

- ▶ `Rectangle3D` hérite de `Rectangle`
- ▶ calcul de la surface pour les `Rectangle3D`  
 $2 * (largeur * hauteur) + 2 * (largeur * profondeur) + 2 * (hauteur * profondeur)$
- ▶ calcul de la surface pour tous les autres `Rectangle` :  
 $(largeur * hauteur)$

## Masquage dans une hiérarchie : exemple simple (2)



La méthode `surface` de `Rectangle3D` **masque** celle de `Rectangle`

- ▶ Un objet de type `Rectangle3D` n'utilisera donc **jamais** la méthode `surface` de la classe `Rectangle`
- ▶ Vocabulaire OO :
  - ▶ Méthode héritée = méthode générale, *méthode par défaut*
  - ▶ Méthode qui masque la méthode héritée = *méthode spécialisée*

## Accès à une méthode masquée

- ▶ Il est parfois souhaitable d'accéder à une méthode/un attribut caché(e)
- ▶ Exemple :
  - ▶ surface des `Rectangle3D` ayant une profondeur nulle (`largeur*hauteur`)
    - ↳ identique au calcul de surface pour les `Rectangle`
- ▶ Code désiré :
  1. Objet non-`Rectangle3D` :
    - ▶ Méthode générale (`surface` de `Rectangle`)
  2. Objet `Rectangle3D` :
    - ▶ Méthode spécialisée (`surface` de `Rectangle3D`)
  3. Objet `Rectangle3D` de profondeur nulle :
    - ▶ D'abord la méthode spécialisée
    - ▶ Ensuite appel à la méthode générale depuis la méthode spécialisée

## Les avatars rencontrent le joueur

```
public static void main(String[] args)
{
    Joueur leJoueur = new Joueur(...);
    Avatar[] lesAvatars = new Avatar[3];

    lesAvatars[0] = new Elfe(...); // Correct ?
    lesAvatars[1] = new Orc(...);
    lesAvatars[2] = new Sorcier(...);

    for (int i=0; i < lesAvatars.length; ++i)
    {
        lesAvatars[i].rencontrer(leJoueur);
    }
}
```



Peut-on mettre un `Sorcier` dans un tableau d'`Avatar` ?

## Accès à une méthode masquée (2)

- ▶ Pour accéder aux attributs/méthodes caché(e)s de la super-classe :

`super + . + variable/méthode`  
`super.rencontrer(leJoueur)`

```
class Rectangle3D extends Rectangle {
    //... constructeurs, attributs comme avant

    public double surface () {
        if (profondeur == 0.0)
            // Acces a la methode masquee
            return super.surface();
        else
            return (2.0*(largeur*hauteur)
                + 2.0*(largeur*profondeur)
                + 2.0*(hauteur*profondeur));
    }
}
```

## Héritage du type des super-classes

Dans une hiérarchie de classes :

- ▶ Un objet d'une sous-classe hérite le type de sa super-classe
- ▶ L'héritage est transitif
- ▶ Un objet peut donc avoir **plusieurs types**

## L'opérateur instanceof

L'opérateur logique `instanceof` permet de **tester le type d'un objet** :

```
Sorcier oz = new Sorcier(...);
boolean b;
b = (oz instanceof Sorcier); // true
b = (oz instanceof Magicien); // true
b = (oz instanceof Avatar); // true
b = (oz instanceof Elfe); // false
```

Il est donc permis d'affecter `Sorcier` à une variable de type `Magicien` ou `Avatar` :

```
Sorcier oz = new Sorcier(...);
Magicien unMagicien = oz; // OK
Avatar unAvatar = oz; // OK
Orc unOrc = oz; // Erreur
```

## Résolution dynamique des liens

Java met en oeuvre le principe de la **"résolution dynamique des liens"**

- ☞ C'est le type effectif et non le type apparent qui est pris en compte

Un petit exemple (cruel mais illustratif) sur les transparents suivants ...

## Choix de la méthode à exécuter

Supposons que la classe `Orc` redéfinisse `rencontrer(Joueur)` et soit le code suivant :

```
Joueur leJoueur = new Joueur(...);
Avatar unAvatar = new Orc(...); // un objet de type Orc
// est affecté à une
// variable de type Avatar
unAvatar.rencontrer(leJoueur);
```

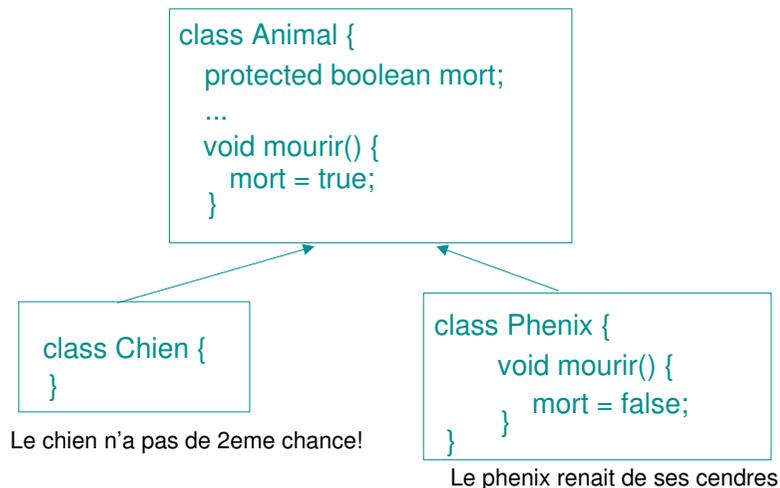
Quelle méthode `rencontrer(Joueur)` va être exécutée ?

En fait, la méthode à exécuter peut être choisie de 2 façons :

1. Résolution *statique* des liens :
  - ▶ Le **type apparent** (type de la variable) est déterminant
  - ▶ `unAvatar` est *déclarée* comme une *variable* de type `Avatar`
  - ▶ Choix de la méthode de la classe `Avatar` (l'avatar salue le joueur)
2. Résolution *dynamique* des liens :
  - ▶ Le **type effectif** (celui de l'objet effectivement stocké dans la variable) est déterminant
  - ▶ `unAvatar` contient un *objet* de type `Orc`
  - ▶ Choix de la méthode de la classe `Orc` (l'orc use de son arme sur le joueur)

## Résolution dynamique des liens - Exemple (1)

Soit la hiérarchie de classes suivante :



## Résolution dynamique des liens - Exemple (2)

Soit maintenant la classe suivante :

```
class Assassin {  
    public tuer(Animal a1, Animal a2) {  
        a1.mourir();  
        a2.mourir();  
    }  
}
```

Que ce passe-t-il lors de l'exécution du code suivant ?

```
Assassin leMechant = new Assassin(...);  
Chien pauvreChien = new Chien(...);  
Phenix pauvrePhenix = new Phenix(...);  
leMechant.tuer(pauvreChien, pauvrePhenix);
```

## Polymorphisme

Les deux ingrédients :

- ▶ héritage du type dans une hiérarchie de classes,
- ▶ et résolution dynamique des liens

permettent de mettre en oeuvre ce que l'on appelle le **polymorphisme**.

- ▶ Un même code s'exécute de façon différente selon la donnée à laquelle il s'applique.
- 🗨 Nous y reviendrons plus en détail au cours prochain

## Résolution dynamique des liens - Exemple (3)

- ▶ Avec la résolution "statique" des liens, dans `Assassin.tuer`, ce serait toujours `Animal.mourir()` qui serait appelé (**c'est le type apparent des variables qui décide**)
  - 🗨 Le phénix meurt comme un vulgaire animal
- ▶ Avec la résolution "dynamique" des liens, dans `Assassin.tuer`, `Animal.mourir()` est appelée pour le chien (car il ne sait mourir que de cette façon) mais `Phenix.mourir()` est appelée pour le phénix (**c'est le type effectif qui décide**)
  - 🗨 Le chien meurt, le phénix survit !
  - 🗨 C'est ce qui va se passer en Java

## Au sujet de instanceof

**Attention :** L'opérateur `instanceof` est à utiliser avec précaution.

- 🗨 Il peut vous amener à ne pas utiliser le polymorphisme là où il faudrait, nous y reviendrons aussi au prochain cours

## Ce que j'ai appris aujourd'hui

- ▶ Que l'on peut réduire la duplication de code et reproduire de bon modèles de la réalité en utilisant des **hiérarchies de classe**
  - ▶ Qu'une sous-classe hérite des membres de ses classes parentes ainsi que de leurs types
  - ▶ Que la construction d'objets dans le cadre de l'héritage obéit à des règles précises (`super(...)` en Java)
  - ▶ Qu'en Java, pour que la notion d'héritage soit établie il faut avoir recours à `extends`
  - ▶ Par quel mécanisme on peut gérer le masquage de variables et de méthodes dans une hiérarchie de classes
  - ▶ Que Java implémente le principe de **résolution dynamique des liens** : le type de l'objet, plutôt que le type de la variable qui référence l'objet détermine la méthode à exécuter
- ☞ Je peux maintenant **améliorer la modularisation** de mes programmes orientés objet en Java

## Notion de paquetage

En Java, il est possible de regrouper les classes en paquetages (bibliothèques). Les paquetages ont une incidence sur la gestion des droits d'accès.

- ☞ On met usuellement dans un même paquetage des classes relatives à un concept commun

Pour les programmes de taille importante auxquels plusieurs programmeurs contribuent, la notion de paquetage permet notamment :

- ▶ une meilleure organisation ;
- ▶ d'éviter des conflits de noms.

Si l'on ne fait rien de particulier, les classes que l'on programme se trouvent de facto dans un paquetage : le paquetage par défaut.



## Héritage



Spécifier un *lien d'héritage* :

```
class SousClasse extends SuperClasse {...}
```

*Droits d'accès* : `protected` accès autorisé au sein de la hiérarchie (et dans toutes les classes du même paquetage)

*Masquage/Shadowing* : un même attribut (ou méthode statique) peut être présent dans une sous-classe et une super-classe  
*Redéfinition/Overriding* : une méthode d'instance peut-être présente dans une sous-classe et une super-classe

Accès à un *membre masqué/redéfini* : `super.membre`

Le constructeur d'une sous classe doit faire appel au *constructeur de la super classe* :

```
class SousClasse extends SuperClasse {  
    SousClasse(liste de paramètres) {  
        super(arguments); // première instruction  
        ...  
        ...  
    }  
}
```

## Nom unique des entités (1)

Java utilise des noms pour identifier les entités du programme : classes, interfaces, etc.

**Ces noms doivent être uniques**, faute de quoi il y a ambiguïté

- ☞ si plusieurs entités ont le même nom, comment savoir laquelle utiliser ?

## Nom unique des entités (2)

Lorsqu'on écrit un programme de A à Z, garantir l'unicité des noms est possible.

Mais que faire lorsqu'on écrit des bibliothèques ayant pour but d'être utilisées par des milliers de programmeurs, dans autant de programmes différents ?

- ↳ la bibliothèque standard de Java 21 comporte plus de 5000 classes et interfaces publiques, donc autant de noms !

Comment garantir l'absence de conflit avec les noms choisis par tous les programmeurs utilisant cette bibliothèque ?

## Conflit de noms : exemple

La notion de table associative (« map » en anglais) est une structure de données très couramment utilisée en programmation.

- ↳ La bibliothèque Java possède ainsi une interface **Map** utilisée de façon extensive

Le mot « map » désigne aussi une carte en anglais

- ↳ Un programme cartographique a donc de très grandes chances de comporter une classe ou une interface nommée également **Map**

- ↳ comment distinguer ces deux utilisations du même nom si un seul programme a besoin des deux ?

## Mauvaise solution : préfixes

Idée : préfixer tous les noms par une chaîne qu'on espère globalement unique.

Par exemple, l'interface **Map** pour les tables associatives pourrait être nommée **CollMap** (fait partie de « Collections » en Java), tandis que la classe **Map** des cartes pourrait être nommée **CartoMap**.

Plusieurs problèmes :

- ▶ ils devraient être longs pour être uniques, mais
  - ▶ ils devraient être courts pour ne pas gêner, et
  - ▶ ils doivent être utilisés même en l'absence de conflit puisqu'ils font partie du nom !
- ↳ Pour tenter de parer à ces problèmes, Java offre la notion de **paquetage** (« packages »)

## Paquetage

Un paquetage est une entité nommée qui contient un certain nombre de types (classes et/ou interfaces).

Au début de chaque fichier source, il est possible de spécifier le paquetage dans lequel placer les classes et interfaces qu'il contient au moyen de l'instruction :

```
package nomDuPaquetage;
```

Exemple :

```
package collections;  
  
public interface Map { ... }
```

## Noms qualifiés

Le **nom complètement qualifié** («fully-qualified name ») d'un type déclaré à l'intérieur d'un paquetage inclut le nom de ce dernier

Exemple :

```
collections.Map
```

est le nom complètement qualifié du type **Map** ainsi déclaré :

```
package collections;  
  
public interface Map {  
    ...  
}
```

Les paquetages jouent donc un **rôle similaire aux préfixes**, avec l'avantage de pouvoir être **omis dans la majorité des cas**.

## Importation

Pour éviter de devoir utiliser la version complètement qualifiée des noms définis dans un autre paquetage, il est possible d'importer les noms utilisés, au moyen de l'énoncé **import**

L'exemple précédent peut s'écrire :

```
package wordprocessor;  
import collections.Map;  
class Dictionary implements Map { ... }
```

## Noms qualifiés

Tous les noms d'un paquetage sont utilisables sans préfixe à l'intérieur de ce même paquetage.

C'est aussi vrai si un paquetage est réparti sur plusieurs fichiers !

```
package collections;  
public class HashMap implements Map { ... }
```

A l'**extérieur** d'un paquetage donné, les noms **publics** de ce dernier sont utilisables en **version totalement qualifiée**.

Exemple :

```
package wordprocessor;  
  
class Dictionary implements collections.Map { ... }
```

## Importation (2)

Il est interdit d'importer deux noms identiques, ou de définir un nom identique à un nom importé :

```
package wordprocessor;  
import collections.Map;  
import cartography.Map; // NON !!
```

Tous les énoncés **import** doivent apparaître juste après l'énoncé **package** (et nulle part ailleurs).

## Imports multiples

Il est aussi possible d'importer la totalité des noms d'un paquetage au moyen de l'astérisque.

Exemple :

```
import java.util.*;
```

- ↳ importe la totalité des noms du paquetage `java.util` (**Map**, **Set**, **Scanner**, et beaucoup d'autres !)
- ↳ Déconseillé, car il devient alors difficile de savoir quel ensemble de noms est importé, et il peut même changer au cours du temps !

## Imports multiples (2)

De nos jours, les environnements de développement comme Eclipse gèrent automatiquement les importations.

## Règles de visibilité

Les paquetages influencent la visibilité des noms :

1. Les types des classes ou interfaces qui ne sont pas déclarés **public** sont visibles uniquement dans le paquetage dans lequel ils sont déclarés.
2. Les membres qui ne sont déclarés ni **public** ni **private** sont visibles dans le paquetage dans lequel leur propriétaire est déclaré.
3. Les membres qui sont déclarés **protected** sont visibles dans le paquetage dans lequel leur propriétaire est déclaré, et dans toutes les sous-classes, indépendamment de leur paquetage

## Hierarchie de paquetages

Les paquetages peuvent être organisés en hiérarchie, c-à-d qu'un paquetage peut contenir d'autres paquetages, et ainsi de suite.

Par exemple, il existe un paquetage standard nommé `java`, dans lequel se trouvent plusieurs sous-paquetages comme `java.lang` et `java.util`

A l'intérieur de ce dernier se trouvent aussi bien des classes et interfaces (p.ex. **List**, **Map**) que d'autres sous-paquetages comme `java.util.concurrent`.

## Répertoires et fichiers

Le contenu d'un paquetage peut être réparti sur plusieurs fichiers. Chacun d'entre-eux doit commencer avec un énoncé `package` approprié.

Les fichiers doivent être placés dans des répertoires **portant le nom du paquetage** .

Par exemple, si l'interface `Map` est déclarée à l'intérieur d'un fichier débutant ainsi :

```
package java.util;  
public interface Map { ... }
```

alors ce fichier doit être stocké dans un répertoire nommé `util`, lui-même placé dans un répertoire nommé `java`.

Et bien entendu, le fichier doit être nommé `Map.java`

## Nommage de paquetages

Utiliser des paquetages ne fait que repousser un peu plus loin le problème de l'unicité des noms ...

☞ Comment éviter que deux programmeurs définissent des paquetages de même nom ?

**Idée** : utiliser le nom de domaine Internet de l'organisation (unique), inversé, comme préfixe du nom de paquetage.

Exemple : le nom de domaine de l'EPFL est `epfl.ch`

☞ Tous les paquetages développés à l'EPFL peuvent commencer par `ch.epfl`

Exemple : `ch.epfl.collections`

Problème : les organisations sont renommées, rachetées (Sun par Oracle, p.ex.), disparaissent, etc.

## Importation statique

Le mot-clef `import` peut aussi être suivi du mot-clef `static` pour importer les noms des membres statiques de classes (ou interfaces) sans devoir nommer à chaque fois la classe (ou l'interface). On nomme cela **importation statique**.

Attention, ce type d'importation n'a rien à voir avec les paquetages !

L'astérisque peut également être utilisée pour importer la totalité des membres statiques.

## Importation statique (2)

Par exemple, la classe `Math` (du paquetage `java.lang`) possède un champ statique nommé `PI` contenant la constante du même nom, et des méthodes comme `sin`, `cos` etc.

Pour utiliser ces membres statiques, on peut bien entendu les préfixer du nom de la classe : `Math.PI`, `Math.sin(...)` etc. Mais il est aussi possible de les importer statiquement puis de les utiliser ensuite sans préfixe :

```
import static java.lang.Math.sin;  
  
class MyClass {  
    double sin2(double x) {  
        return sin(x) * sin(x);  
    }  
}
```