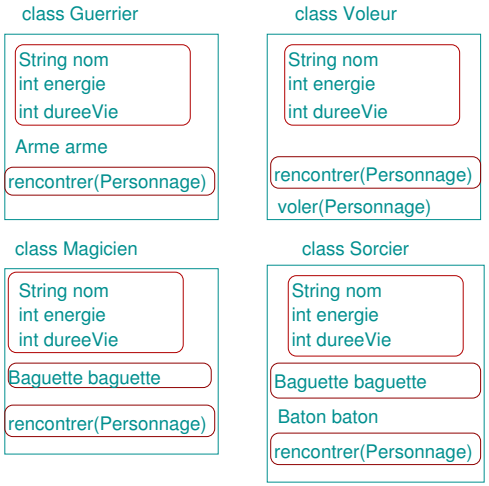
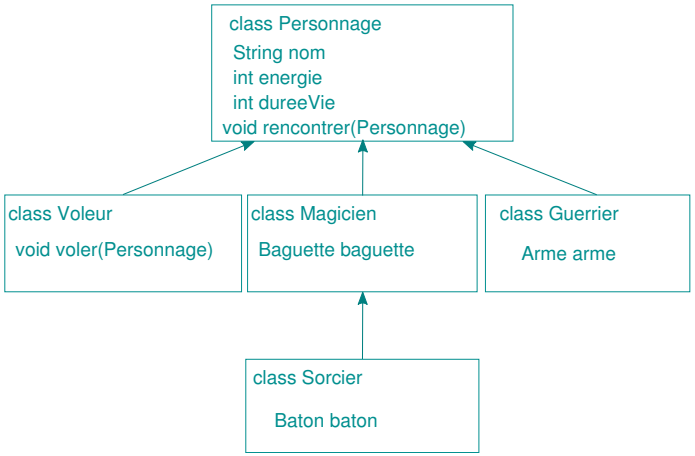
Un exemple

Oublions un peu les rectangles ...

Exemple : classes pour les personnages



Exemple : héritage

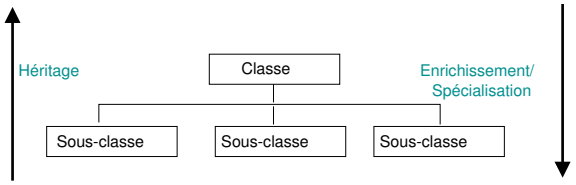


Héritage

Après les notions d'*encapsulation* et d'*abstraction*, le troisième aspect essentiel de la « Programmation Orientée Objet » est la notion d'**héritage**.

L'héritage représente la relation «**est-un**».

Il permet de créer des classes *plus spécialisées*, appelées **sous-classes**, à partir de classes plus générales déjà existantes, appelées **super-classes**.



Héritage (2)

Lorsqu'une sous-classe `C1` est créée à partir d'une super-classe `C`,

- le type est *hérité* : un `C1` **est** (aussi) **un** `C`
- `C1` va *hériter* de l'ensemble :
  - des attributs de `C`
  - des méthodes de `C`
    (sauf les constructeurs)

  ☞ Les attributs et méthodes de `C` vont être disponibles pour `C1` sans que l'on ait besoin de les redéfinir explicitement dans `C1`.
- Par ailleurs :
  - des attributs et/ou méthodes supplémentaires peuvent être définis par la sous-classe `C1`
    ☞ **enrichissement**
  - des méthodes héritées de `C` peuvent être redéfinies dans `C1`
    ☞ **spécialisation**

Héritage : exemple

Lorsqu'une sous-classe `C1` (ici `Guerrier` ou `Voleur`) est créée à partir d'une super-classe `C` (ici `Personnage`),

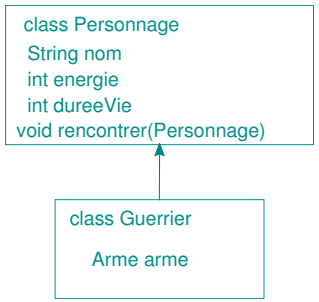
- le type est *hérité* : un `Guerrier` **est** (aussi) **un** `Personnage` :

```
Personnage p;
Guerrier g;
// ...
p = g;
// ...
void afficher(Personnage);
// ...
afficher(g);
```

Héritage : exemple

Lorsqu'une sous-classe `C1` (ici `Guerrier` ou `Voleur`) est créée à partir d'une super-classe `C` (ici `Personnage`),

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




```
Guerrier g = new Guerrier(...);
Voleur   v = new Voleur(...);

g.rencontrer(v);
//...
// dans une méthode de Guerrier
   energie = //...
```

Héritage : exemple

Lorsqu'une sous-classe `C1` (ici `Guerrier` ou `Voleur`) est créée à partir d'une super-classe `C` (ici `Personnage`),

- des attributs et/ou méthodes supplémentaires peuvent être définis par la sous-classe `Guerrier` : `arme`
- des méthodes héritées de `Personnage` peuvent être redéfinies dans `Voleur` : `rencontrer(Personnage)`

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")
- il ne doit **jamais** décrire une relation « a-un »/« possède-un » ("has-a")

Transitivité de l'héritage

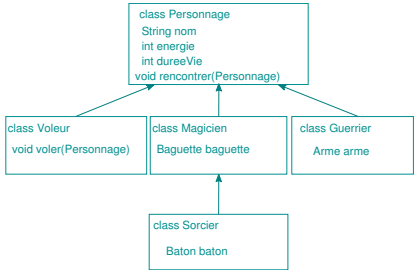
Par transitivité, les instances d'une sous-classe possèdent :

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

Enrichissement par héritage :

- crée un *réseau de dépendances* entre classes,
- ce réseau est organisé en une *structure arborescente* où chacun des nœuds hérite des propriétés de l'ensemble des nœuds du chemin remontant jusqu'à la racine.

☞ ce réseau de dépendances définit une **hiérarchie de classes**



Sous-classe, Super-classes

Une super-classe :

- est une classe « parente »
- déclare les attributs/méthodes communs
- peut avoir plusieurs sous-classes

Une sous-classe est :

- une classe « enfant »
- étend **une seule** super-classe
- hérite des attributs, des méthodes et du type de la super-classe

Un attribut/une méthode hérité(e) peut s'utiliser comme si il/elle était déclaré(e) 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

Passons à la pratique...

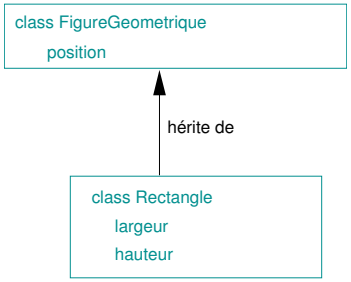
Définition d'une sous-classe en Java :

Syntaxe :

```
class NomSousClasse extends NomSuperClasse
{
 /* Déclaration des attributs et méthodes
    spécifiques à la sous-classe */
}
```

Exemple :

```
class Rectangle extends FigureGeometrique
{
   private double largeur;
   private double hauteur;
   // ...
}
```



Pratique : exemple 2

```
class Personnage {
  // ...
}
// ...
class Guerrier extends Personnage {
  private Arme arme;
  // constructeurs, etc.
}
```

Droit d'accès `protected`

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 exercice)

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 les autres classes du même paquetage). Le mot clé est «`protected`».

Accès protégé et paquetages

Accès protégé (1)

- 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

- ☞ Les attributs protégés sont d'un usage peu recommandé en Java

Accès protégé (2)

Le niveau d'accès protégé correspond à une **extension du niveau privé** permettant l'accès aux sous-classes (et aux autres classes du même paquetage).

Exemple :

```
class Personnage {
// ...
   protected int energie;
}

class Guerrier extends Personnage {
  // ...
   public void frapper(Personnage lePauvre) {
      if (energie > 0) {
    // frapper le perso
   }
}
```

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)

Les `Guerrier` font bande à part

- Pour un personnage non-`Guerrier`:

```
public void rencontrer(Personnage unPersonnage) { saluer(lePersonnage); }
```

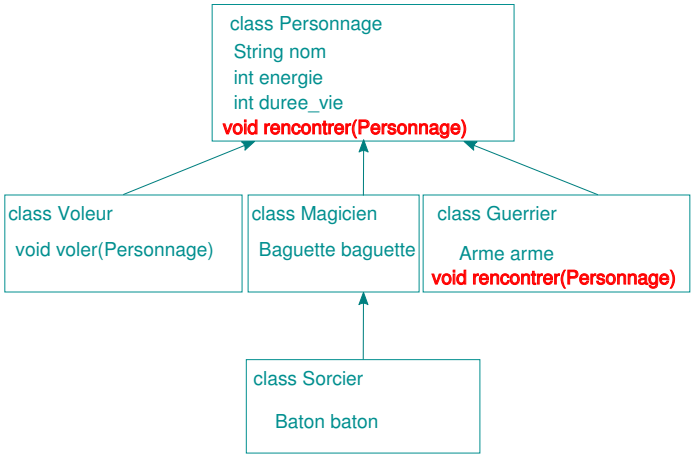
- Pour un `Guerrier`

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

Faut-il re-concevoir toute la hiérarchie ?

☞ Non, on ajoute simplement une méthode `rencontrer(Personnage)` spéciale dans la sous-classe `Guerrier`

Les `Guerrier` font bande à part : masquage/redéfinition

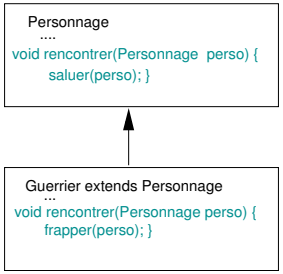


Masquage/Redéfinition dans une hiérarchie

Masquage : pour les variables (« *shadowing* »)
Redéfinition : pour les méthodes (« *overriding* »)

- Masquage : un identificateur qui en cache un autre
- Redéfinition : une méthode déjà définie dans une super-classe a une nouvelle définition dans une sous-classe
- Situations possibles dans une hiérarchie :
  - Même nom d'attribut ou de méthode utilisé sur plusieurs niveaux
  - Peu courant pour les attributs
  - Très courant et **pratique** pour les méthodes

Masquage /Redéfinition dans une hiérarchie (2)



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 OO :
  - Méthode héritée = méthode générale, *méthode par défaut*
  - Méthode qui redéfinit la méthode héritée = *méthode spécialisée*

Accès à une méthode masquée

- Il est parfois souhaitable d'accéder à une méthode/un attribut masqué(e)
- Exemple :
  - Le `Guerrier` commence par rencontrer le personnage comme le fait n'importe quel personnage (il le salue) avant de le frapper !
- Code désiré :
  1. Personnage non-`Guerrier` :
     - Méthode générale (`rencontrer` de `Personnage`)
  2. Personnage `Guerrier` :
     - Méthode spécialisée (`rencontrer` de `Guerrier`)
     - Appel à la méthode générale depuis la méthode spécialisée

Accès à une méthode masquée (2)

Pour accéder aux attributs masqués et aux méthodes redéfinies de la super-classe :

- on utilise le mot réservé `super`
- Syntaxe : `super.` *méthode* ou *attribut*
- Exemple :

```
class Guerrier extends Personnage  {
  //...
  public void rencontrer (Personnage perso) {
    super.rencontrer(perso); // salutation d'usage !!
    frapper(perso);
  }
}
```

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 pourrait 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 : appel explicite

L'invocation du constructeur de la super-classe se fait au tout début du corps du constructeur au moyen du mot réservé `super` .

Syntaxe :

```
SousClasse(liste de paramètres)
{
     /* Arguments : liste d'arguments attendus par
      * un des constructeurs de la super-classe de SousClasse
      */
    super(Arguments);
    // initialisation des attributs de SousClasse ici
}
```

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

☞ le compilateur se charge de réaliser l'invocation du constructeur par défaut

Constructeurs et héritage : exemple 1

Si la classe parente n'admet pas de constructeur par défaut, l'**invocation explicite** d'un de ses constructeurs **est obligatoire** dans les constructeurs de la sous-classe

☞ La sous-classe doit admettre *au moins un constructeur explicite*.

Exemple :

```
class FigureGeometrique {
  private Position position;
  public FigureGeometrique(double x, double y) {
    position = new Position(x,y);
  }
  // ...
}
class Rectangle extends FigureGeometrique {
  private double largeur;
  private double hauteur;
  public Rectangle(double x, double y, double l, double h) {
    super(x,y);
    largeur = l;  hauteur = h;
  } // ...
}
```

Constructeurs et héritage : exemple 2

Autre exemple (qui ne fait pas la même chose) :

```
class FigureGeometrique {
   private Position position;
   public FigureGeometrique() { position = new Position(0.0, 0.0); }
}
class Rectangle extends FigureGeometrique {
   private double largeur;
   private double hauteur;
   public Rectangle(double l, double h) {
      largeur = l;
      hauteur = h;
   }
  // ...
}
```

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 manipulateurs, il
     faudrait alors sûrement aussi les redéfinir) */
}
```

Constructeurs et héritage : résumé (1)

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

Constructeurs et héritage : résumé (2)

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

- Appel automatique à `super()`
- Pratique parfois, mais erreur si le constructeur par défaut n'existe pas

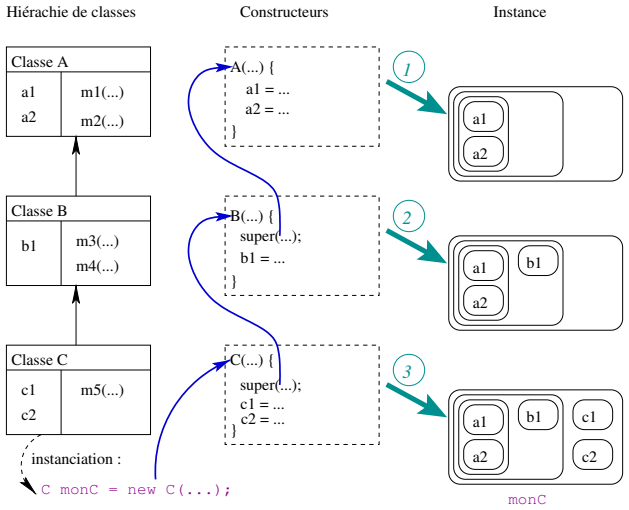
Rappel : le constructeur par défaut est particulier

- 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, dans un premier temps :

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

Ordre d'appel des constructeurs



Les personnages rencontrent le personnage courant

```
public static void main(String[] args)
{
   Personnage   lePersonnage = new Personnage(...);
   Personnage[] personnages  = new Personnage[3];

   personnages[0] = new Voleur(...); // Correct ?
   personnages[1] = new Guerrier(...);
   personnages[2] = new Sorcier(...);

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

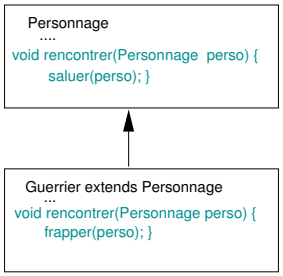
☞ Peut-on mettre un `Sorcier`, un `Voleur` ou un `Guerrier` dans un tableau de `Personnage` ?

Héritage du type : rappel

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**

Choix de la méthode à exécuter (1)



Que fait le code suivant :

```
// ...
Personnage unPersonnage = new Guerrier(...);
unPersonnage.rencontrer(unAutrePersonnage);
```

☞ Quelle méthode `rencontrer(Personnage)` va être exécutée ?

Choix de la méthode à exécuter (2)

```
// ...
Personnage unPersonnage = new Guerrier(...);
unPersonnage.rencontrer(unAutrePersonnage);
```

1. Résolution *statique* des liens :
   - Le type apparent (type de la variable) est déterminant
   - `unPersonnage` est *déclarée* comme une *variable* de type `Personnage`
   - Choix de la méthode de la classe `Personnage` (le personnage salue le personnage courant !)
2. Résolution *dynamique* des liens :
   - Le type effectif (celui de l'objet effectivement stocké dans la variable) est déterminant
   - `unPersonnage` contient la référence à un objet de type `Guerrier`

Résolution dynamique des liens

Java met en œuvre le principe de « **résolution dynamique des liens** »

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

Résolution dynamique des liens – Exemple (1)

```
class Jeu {
  private Personnage joueur;
  private Personnage[] adversaires;
  // ..
  public void tourDeJeu() {
    for (int i = 0; i < adversaires.length; ++i)
    {
      adversaires[i].rencontrer(joueur);
    }
  }
  // ...
}
```

Que se passe-t-il si :

```
adversaires[0] = new Sorcier(...);
adversaires[1] = new Guerrier(...);
// ...
leJeu.tourDeJeu();
```

Résolution dynamique des liens – Exemple (2)

- Avec la résolution « statique » des liens, dans `tourDeJeu`, ce serait toujours `rencontrer(Personnage)` de `Personnage` qui serait appelé (**c'est le type apparent des variables qui décide**)
  - ☞ Le personnage principal du jeu se fait saluer deux fois : une fois par le guerrier et une autre fois par le sorcier
- Avec la résolution « dynamique » des liens, dans `tourDeJeu`, `rencontrer(Personnage)` de `Personnage` est appelée pour le sorcier mais `rencontrer(Personnage)` de `Guerrier` est appelée pour le guerrier (**c'est le type effectif qui décide**)
  - ☞ Le personnage principal du jeu se fait saluer par le sorcier … mais frapper par le guerrier !
  - ☞ C'est ce qui va se passer en Java

Polymorphisme

Les deux ingrédients :

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

permettent de mettre en œuvre 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