Programmation objet

Cours n°4 : Héritage en Java

Alain Giorgetti

giorgett@lifc.univ-fcomte.fr
http://lifc.univ-fcomte.fr/PEOPLE/giorgetti/

Laboratoire d'Informatique
de l'Université de Franche-Comté

Pourquoi hériter ?

- Pour relier des classes entre elles
  - lorsqu'une classe étend les fonctionnalités d'une autre classe
  - lorsqu'une classe adapte le fonctionnement d'une autre classe à une situation particulière
- Pour exploiter des classes existantes
  - sans en modifier le code
  - même si ce code est inaccessible
  - la documentation de la classe héritée suffit
- Pour organiser le développement
  - regrouper les champs communs à plusieurs classes
  - regrouper les méthodes communes
- Pour des applications plus évolutives
  - moins de programmation, plus de conception

Relation d'extension

- Relation entre deux classes
  - ♦ La classe B étend la classe A
    - ➢ tout objet de la classe B est un objet de la classe A
    - ➢ l'ensemble B est inclus dans l'ensemble A
  - ♦ Relation non symétrique
- Motivations de cette extension
  - ♦ Spécialisation
    - ➢ exprime qu'une classe est un cas particulier d'une autre classe
    - ➢ *une préfecture est une ville particulière*
  - ♦ Enrichissement
    - ➢ ajoute des propriétés et/ou des fonctionnalités à une classe
    - ➢ *une préfecture est une ville qui, de plus, abrite un préfet*

Comment déclarer une extension

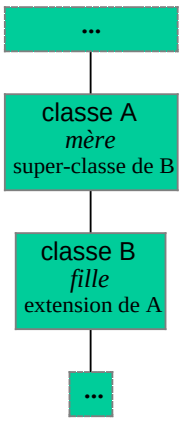
```
class Fille extends Mere {
    … ClasseDuChamp champEnPlus;

    … methodeRedefinie(…) {
    }
    … methodeEnPlus(…) {
    }
}
```

Les objets de la classe ***Fille*** ...

- ♦ héritent des champs et méthodes de la classe ***Mere***
- ♦ disposent de champs supplémentaires
- ♦ redéfinissent certaines méthodes non statiques de ***Mere***
- ♦ disposent de méthodes supplémentaires

Extension et héritage



- La classe B *étend* la classe A
  - La classe B est **une** *classe fille* (ou *extension*) de la classe A
  - La classe A est **une** *classe mère* de B

  Héritage simple : une seule classe mère, appelée *super-classe*
- Relation d'héritage (globale)
  - La classe B peut également avoir des classes filles, qui *héritent* de B et de A
  - On parle de *sous-classes* de B et de A

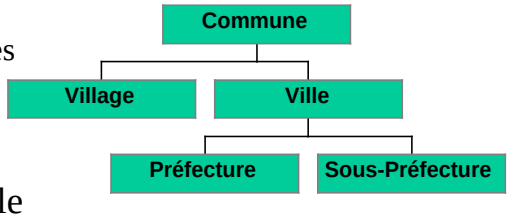
    Les sous-classes de A sont les classes filles de A et leurs sous-classes

Concevoir des extensions

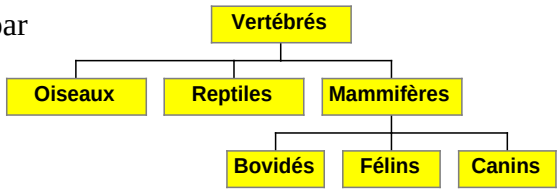
- Repérer les relations « EstUn(e) » entre classes
  - *une préfecture* est une *ville particulière*
- Etendre dans les 2 directions
  - par des sous-classes pour chaque cas particulier connu
  - par des super-classes pour une généralisation potentielle
    - *une ville est une commune particulière*
- Dessiner le graphe d'héritage
  - synonyme : *hiérarchie de classes*
  - représente la relation d'extension par un ensemble d'arbres de classes (*arborescence*)

Hiérarchie de classes : exemples

- Types de communes
  - ♦ propriétés différentes
  - ♦ ajout de champs



- Classification animale
  - ♦ identification par propriétés



- Autres exemples
  - ♦ figures géométriques
  - ♦ classes de nombres

Fonctionnement de l'héritage

- Un objet hérite des champs et des méthodes de toutes ses classes ascendantes
  - ♦ champs hérités
  - ♦ méthodes héritées
- De plus, un objet dispose des champs et des méthodes de sa propre classe (nouveaux noms)
  - ♦ champs supplémentaires
  - ♦ méthodes supplémentaires
- Dans une classe, on peut redéfinir une méthode d'une classe ascendante
  - ♦ méthodes redéfinies

Champs hérités

- Fonctionnement
  - ♦ Comme si un objet possédait tous les champs de toutes ses classes ascendantes, en plus de ses propres champs
  - ♦ Tous les noms de champs d'une lignée doivent être différents
- Syntaxe
  - ♦ la même que pour les champs propres

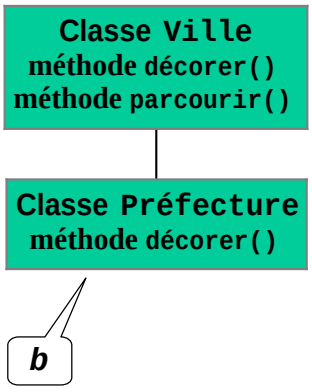
```
unObjet.champPropre;
unObjet.champAscendant;
```

  - ♦ Pour **`this`** dans une méthode, même syntaxe

```
this.champPropre;
this.champAscendant;
```


Méthodes héritées et redéfinies

- ♦ On a construit un objet **`b`** de classe **`Préfecture`**
- ♦ Appel : **`b.parcourir();`**
- ♦ Méthode exécutée
  - ➢ méthode **`parcourir()`** non définie dans la classe **`Préfecture`** de l'objet appelant **`b`**
  - ➢ méthode cherchée et trouvée dans la classe ascendante **`Ville`**
- ♦ Appel : **`b.décorer();`**
- ♦ Méthode redéfinie
  - ➢ méthode **`décorer()`** de la classe **`Préfecture`** exécutée



Méthodes redéfinies

- Définition

  Une méthode est *redéfinie* si elle porte **le même nom** et si elle a la **même liste de paramètres** que dans l'une de ses classes ascendantes

- Exemple : *décorer une ville, décorer une préfecture*
- Appel d'une méthode dans la super-classe
  - ➢ mot réservé **`super`**

```
class Préfecture {
  void décorer() {
    super.décorer();
    this.batiment.setDrapeau();
  }
}
```

- *méthode* ***surchargée*** : même nom, mais paramètres différents

Héritage et constructeurs

- Un objet de classe **B** est construit par une instruction **`new B(…)`**
- La classe **B** hérite de la classe **A**, donc :
  - ♦ un constructeur de la super-classe **A** doit être appelé
  - ♦ Règle d'appel des constructeurs : la construction d'un objet de classe **B** commence toujours par l'appel d'un constructeur de la super-classe de **B**
- Cas des constructeurs implicites
  - ♦ Quel est le constructeur de **A** appelé ?

Constructeur de la super classe appelé

- Dans le corps d'un constructeur explicite **`B(…),`** la première instruction est :
  - ♦ une instruction **`super(…)`** : invoque le constructeur (choisi par son profil) de la super-classe **`A`**
  - ♦ une instruction **`this(…)`** : invoque un autre constructeur (choisi par son profil) de la classe **`B`**
  - ♦ ni **`this(…)`**, ni **`super(…)`**: le compilateur introduit un appel à **`super()`**
- Conclusions
  - ♦ pour l'héritage, toujours s'assurer que le constructeur sans paramètres existe, en l'explicitant
  - ♦ organiser les constructeurs en pyramide

Classe finale

- Classe finale : classe qui ne peut pas être super-classe d'une autre classe
  - ♦ Syntaxe

```
final class ...
```

  - ♦ Sémantique
    - ➢ Si un objet est de classe déclarée A, alors sa classe d'exécution ne peut être que A
  - ♦ Par défaut : non final

Méthode finale

- Méthode finale : Une sous classe ne pourra pas redéfinir la méthode
  - ♦ Syntaxe dans le corps d'une classe (disons **A**)

```
final … méthode(…) {
}
```

  - ♦ Sémantique : on fixe une fois pour toutes un comportement, pour les objets de classe **A**
  - ♦ Par défaut : non final

Champ final

- Champ final : ni la classe, ni aucune sous classe ne peuvent changer la valeur initiale du champ
  - ♦ Joue le rôle d'une constante d'objet ou de classe
  - ♦ Syntaxe

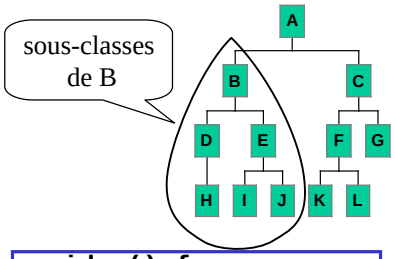
```
final typeChamp nomChamp = valInitiale;
```

  - ♦ Sémantique : on fixe une fois pour toutes la valeur de ce champ
  - ♦ Par défaut : non final

Transtypage entre classes

Une classe est *compatible* avec toutes ses sous-classes

- Affectation entre objets : la partie droite de l'affectation doit être compatible avec la partie gauche
- Même règle pour la transmission d'arguments réels aux paramètres formels d'une méthode ou d'un constructeur
- Mécanisme de transtypage (*cast*) implicite
- Erreur : `incompatible type for =, explicit cast needed`




```
void m() {
  B objB = new B(…);
  I objI = new I(…);
  A objA = new A(…);
     …
  objB = objI; //OK
  objB = objA; //faux
}
```


Classe déclarée et classe réelle

- Classe déclarée
  - Chaque variable est déclarée (définitivement) d'une classe donnée : c'est sa ***classe déclarée***
- Classe réelle (*runtime class*)
  - Certaines instructions changent la classe d'une variable
  - Au cours de l'exécution, une variable peut avoir plusieurs ***classes réelles*** (ou *classes d'exécution*)
- Règle : la classe réelle hérite de la classe déclarée
- Mot réservé `instanceof`

  `var instanceof Classe`

  vrai si et seulement si la classe réelle de la variable hérite de ***`Classe`***

Changer la classe d'une variable

- Variable : *objet* ou *champ* ou *paramètre formel* de classe **A**

```
A var;
```

- Instructions modifiant la classe de la variable **var**
  - construction d'un sous-objet (objet d'une sous-classe **B** de **A**)

```
var = new  B(…);
```

  - affectation d'un objet d'une sous-classe

```
B obj = new B(…);
var = obj;
```

  - passage de paramètre

```
… méthode(A var) {
    …
}
```

```
… autreMéthode(…) {
   B obj;
   … méthode(B);
}
```


Transtypage explicite : exemple

classe **Date**
`Date(int j, int m)`

classe **Ferie**
`Ferie(int j, int m, String s)`
`String nomFete()`

```
void repos() {
  Date d;
  Ferie f;

  d = new Ferie(25,12,"Noel");
    …
  if (d instanceof Ferie) {
    f = (Ferie) d;
    System.out.print(
      f.nomFete()
    );
  }
}
```

classe réelle de **d**, personnelle

classe déclarée de **d**, inaccessible

Transtypage (cast) explicite

- Syntaxe `(NomClasse) obj;`
  - Effet : change la classe d'exécution
  - Condition : la classe d'exécution de **`obj`** doit hériter de **`NomClasse`**
  - Sinon, une exception **`ClassCastException`** est levée
- Solution
  - le mot réservé **`instanceof`**
- Exemple

```
Ville v;
Préfecture p;  // sous-classe de Ville
   // construction ou affectation de v
if (v instanceof Préfecture) {
    p = (Préfecture) v;
}
```


Connaître la classe d'exécution

- La classe **`Object`**
  - toutes les classes Java en héritent, directement ou non
  - elle dispose d'une méthode

```
public final Class getClass()
```

    - retourne la classe d'exécution (*runtime class*) de l'objet appelant
    - Cette classe est elle-même un objet de la classe **`Class`**
  - La classe **`Class`** dispose d'une méthode

```
public String getName()
```

- Conclusion
  - Si **`obj`** est un objet, le nom de sa classe d'exécution est :

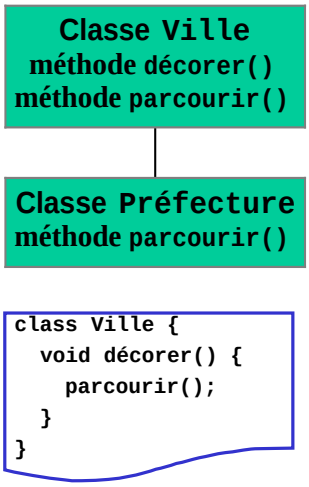
```
obj.getClass().getName()
```

Méthode d'objet et classe réelle

- Appel

```
b.décorer();
```

- Effet
  - La méthode **`décorer()`** appelle la méthode **`parcourir()`**
  - or, la classe réelle de **`b`** est **`Préfecture`**
  - C'est la méthode **`parcourir()`** de la classe **`Préfecture`** qui est appelée



Usages de l'héritage

- ***Polymorphisme*** : traitement analogue d'objets de classes différentes
  - Pour mémoriser facilement les noms des méthodes
  - Pour permettre la liaison dynamique
  - Exemple : **`afficher()`**, **`entrer()`**, …
- ***Liaison dynamique*** : mécanisme automatique de choix d'une méthode en fonction de la classe d'exécution de l'objet appelant
- Conclusions
  - Des classes hiérarchisées réduisent le travail de programmation
  - La classe d'un objet fournit de l'information
    - sur cet objet
    - sur les traitements que cet objet peut subir