Bases de la POO / Java

1

La relation d'héritage

CLASSIFICATION PAR HÉRITAGE
L'HÉRITAGE, UNE RELATION ENTRE CLASSES
CARACTÉRISTIQUES DE L'HÉRITAGE

Une relation entre classes

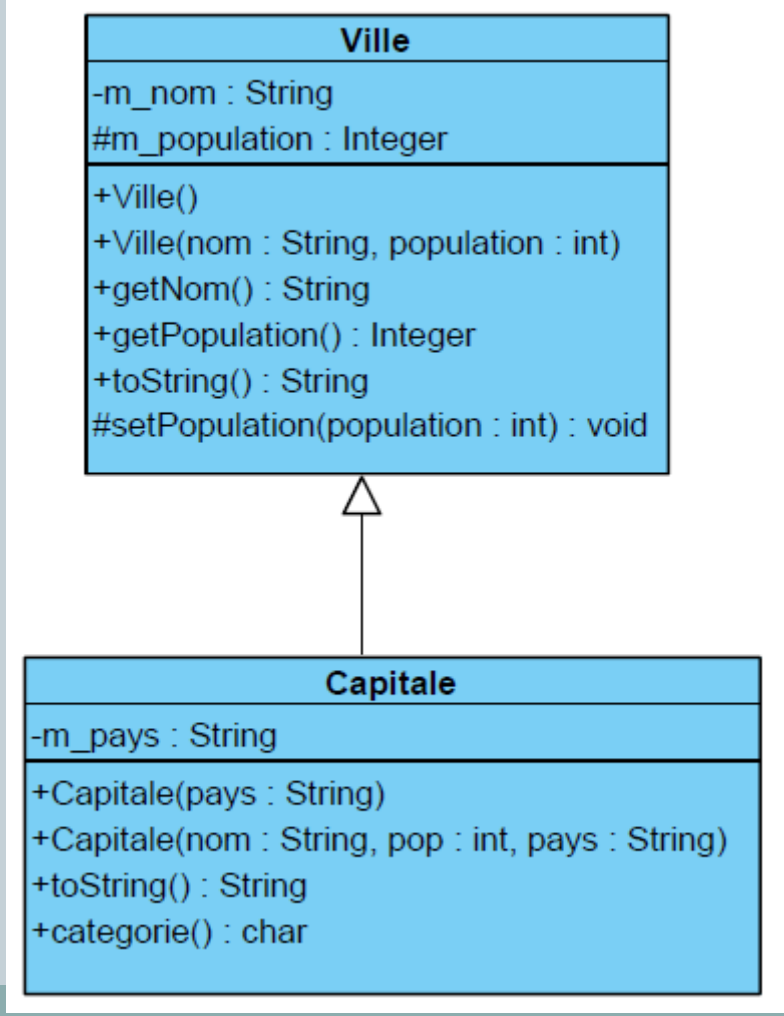
➔ Définition

Une classe **C2 hérite** d'une classe C1 si elle se décrit sémantiquement comme une **spécialisation** de cette dernière.

- **Toute propriété de C1 est une propriété de C2**
- **Tout comportement de C1 est un comportement de C2, avec une éventuelle redéfinition**
- **C2 peut décrire de nouvelles propriétés**
- **C2 peut modéliser de nouveaux comportements**

Représentation UML

spécialisation



généralisation

Classification par héritage (1)

➔ Origine et justification

- Principes de la classification en S.V.T.
- Factorisation hiérarchique des caractéristiques
- Arbre n-aire de classification
- Exemples :
  - Invertébrés (pas de colonne vertébrale)
  - Insectes (1 paire antennes et 3 paires de pattes)
  - ---

Classification par héritage (2)

➔ Objectifs recherchés pour la relation "is_a"

- Favoriser la réutilisation de codes sources
- Réutilisation des données et des traitements
- Respecter les règles d'encapsulation
- Favoriser la conception par spécialisation progressive
  - Spécialisation par enrichissement
  - Spécialisation par substitution (redéfinition)

Classification par héritage (3)

→ Vocabulaire attaché à la relation "is_a"

- Relation exclusivement entre classes
- Classe amont : **classe mère**
- Classe aval : **classe fille (classe dérivée)**
- Relation hiérarchique à plusieurs niveaux
- Ensemble des classes aval : famille
- Transitivité restreinte

Caractéristiques de l'héritage

- **Simple (pas d'héritage multiple)**
- **Seuls les constructeurs ne sont pas hérités**
- **Héritage possible**
  - Si classe C1 visible par C2
    - `public class C1` ou même package
  - Si classe C1 non «finale»
    - finale = n'est pas dérivable
    - `final class C1`

La classe mère (1)

➔ Les attributs

- Attributs publics (**public**), privés (**private**) et délégués (**protected**)
- Les attributs publics sont accessibles par toute classe
- Les attributs privés restent encapsulés par la mère
- Les attributs délégués sont accessibles directement par la fille*
- Délégation dans la descendance (nue propriété en droit)

* et par les autres classes d'un même package

La classe mère (2)

➔ Les méthodes

- L'usage de toutes les méthodes est exporté
- Byte codes partagés entre mère et filles
- Spécificateur d'accès **protected** pour en restreindre l'usage à la seule descendance
- Possibilité de redéfinition et de surcharge par les filles

Exemple de Classe mère : Ville


```
public class Ville
{
   private String nom;
   protected int population;

   public Ville ()
   {
         nom = "NICE";
         population = 345000;
   }

   public Ville (String nom, int pop)
   {
      nom   = nom.toUpperCase();
      population = pop;
   }

   public String  getNom    () {return nom;}
   public int getPopulation () {return population;}
   protected void setPopulation (int pop) {population=pop;}

   public String toString() {return nom + ", " + pop;}

}
```

- Nombre d'attributs : 2
- Nombre de constructeurs : 2
- Nombre de méthodes : 4

La classe fille (1)

- Attributs
  - Attributs privés hérités de la classe mère
  - Attributs délégués par la classe mère (**protected**)
  - Hérités de la mère et surchargés par la fille
    - Même nom, mais type différent
  - Attributs spécifiques à la fille
- Seuls les 2 derniers doivent être déclarés dans la classe fille

Définition d'une classe fille

➔ **utilisation du mot-clé extends**

```
public class CF extends CM
{
    // …

}
```

Exemple de classe fille : Capitale


```
public class Capitale extends Ville
{
   // attributs spécifiques de Capitale
   private String pays;

   // …
}
```

La classe fille (2)

➔ Les constructeurs

- Spécifiques à chaque classe fille
- Invoquer un constructeur de la mère (**super**), pour sous-traiter l'initialisation des attributs privés hérités

Constructeur d'instances de CF

➔ **Syntaxe Java**

```
public CF (list params)
{
    super (sub list params);
    // Affectation des attributs propres à CF
    …
}
```

La classe fille (3)

- Méthodes
  - Héritées de la mère
  - Héritées de la mère et **redéfinies** par la fille
    - Même nom, même signature (type et nombre de paramètres, valeur de retour et exceptions propagées doivent être identiques)
  - **Surchargées**
    - Même nom, mais signature différente
    - ⇒ C'est une nouvelle méthode propre à la fille
  - Méthodes spécifiques à la fille
  - Possibilité d'appeler une méthode de la mère (**super**)
- Les 3 dernières doivent être déclarées et implémentées dans la classe fille

Constructeurs et méthodes de Capitale


```
public class Capitale extends Ville
{    // attribut spécifique
    private String pays;

    // constructeurs
    public Capitale(String pa) {super(); pays = pa.toUpperCase();}
    public Capitale(String nom, int pop, String pa)
    {
         super(nom, pop);
         pays = pa.toUpperCase();
    }

    // méthode redéfinie
    public String toString()
    {
         return super.toString() + ", Pays : " + pays ;
    }

    // méthode spécifique
    public char categorie()
    {
          return 'C';
    }
}
```

Bilan

➔ **Classe Capitale**

- Attributs : **3** (2 + 1)
- Constructeurs : 2
- Méthodes : **5** (4+1)

➔ **Classe Ville**

- Attributs : 2
- Constructeurs : 2
- Méthodes : 4

Tests du constructeur normal


```
public class Test_Capitale
{
   public static void main (String[] args)
   {
      Tests.Begin("Capitale", "1.0.0");
         Tests.Design("Controle des constructeurs",3);
         {
            Tests.Case ("Constructeur normal");
            {
Capitale Paris   = new Capitale("Paris", 2234105, "France");
Tests.Unit("PARIS, 2234105, Pays : FRANCE", Paris.toString());
            }
         }
      Tests.End();
   }
}
```

Tests du premier constructeur


```
public class Test_Capitale
{
   public static void main (String[] args)
   {
      Tests.Begin("Capitale", "1.0.0");
         Tests.Design("Controle des constructeurs",3);
         {
            Tests.Case ("Constructeur normal");
            {
Capitale x = new Capitale("Italie");
Tests.Unit("NICE, 345000, Pays : ITALIE", x.toString());
            }
         }
      Tests.End();
   }
}
```

Héritage implicite

➔ **Langage Java**

Toute classe est implicitement la fille d'une classe prédéfinie **Object** du package java.lang.

Il n'y a pas de véritable sémantique attachée à cet héritage.

Cet héritage fourni une définition préalable et par défaut des méthodes ***toString***, ***equals***, ***clone*** et ***getClass*.**

Run Time Type Identification

Possibilité de déterminer dynamiquement la classe d'origine d'un objet à partir de sa seule référence.

Accéder à la "carte d'identité" de l'objet cible

Accéder au RTTI

Via l'opérateur **instanceof**

Via la séquence : **getClass().getName()**

Accès au RTTI (1)

Cas 1 : classe disponible à la compilation

```
if (o instanceof LinkedHashMap) i++;
```

o est une référence sur un objet existant.

Second opérande contrôlé à la compilation !

Accès au RTTI (2)

Cas 2 : classe indisponible à la compilation

Usage de la méthode **getClass** de Object

String w= o.**getClass()**.**getName**();

if (w.equals("java.util.HashMap")) i++;

o est une référence sur un objet existant.

Aucun contrôle à la compilation !

Transtypage implicite (1)

➔Définition

Capacité d'une variable de type CM à recevoir une référence sur un objet de sa descendance.

- **Dynamique**
- **Implicite**
- **Ascendant**

Transtypage implicite (2)


```
//…
Capitale Paris =
   new Capitale("Paris",2250000,"France");
// Capitale hérite de Ville
Ville v = Paris;
Tests.Unit("PARIS, 2250000, Pays : FRANCE",
    v.toString());

// Capitale hérite de Ville, qui hérite d'Object
Object o = Paris;
```

Intérêt


```
//…
Ville[] tabVille = new Ville[10];
tabVille[0] = new Ville("Nice",100);
tabVille[1] = new Capitale("Paris", 200, "France");

for (int i=0; i<2; i++)
      System.out.println(tabVille[i].toString());
```

Transtypage explicite


```
public static void main (String[] args)  {
Capitale c1= new Capitale (...);
Ville v= c1;
Capitale c2;
        c2= (Capitale) v;
}
```

Polymorphisme

- **Polymorphisme = une méthode peut avoir plusieurs formes**
- Deux types de polymorphismes
- Polymorphisme statique = surcharge
  - Méthodes avec même nom mais signatures différentes
  - Détection de la bonne méthode à la compilation
  - Ex : constructeurs d'une classe
- Polymorphisme dynamique = redéfinition
  - Une sous-classe redéfinit une méthode d'une classe mère
  - Détection de la bonne méthode à l'exécution
  - Ex : méthode toString de Ville rédéfinie dans Capitale

Utilisation de final

- Pour empêcher la redéfinition d'une méthode
- Une méthode **final** ne peut pas être redéfinie
- Conseil : déclarer les accesseurs en méthodes finales
  - `public final String getNom() {return m_nom;}`