Table des matières

L'héritage

Je vous arrête tout de suite, vous ne toucherez rien. Pas de rapport d'argent entre nous ! Non, la notion d'héritage en programmation est différente de celle que vous connaissez, bien qu'elle en soit tout de même proche. C'est l'un des fondements de la programmation orientée objet !

Imaginons que, dans le programme réalisé précédemment, nous voulions créer un autre type d'objet : des objets Capitale. Ceux-ci ne seront rien d'autre que des objets Ville avec un paramètre en plus : un monument. Vous n'allez tout de même pas recoder tout le contenu de la classe Ville dans la nouvelle classe ! Déjà, ce serait vraiment contraignant, mais en plus, si vous aviez à modifier le fonctionnement de la catégorisation de nos objets Ville, vous auriez aussi à effectuer la modification dans la nouvelle classe… Ce n'est pas terrible.

Heureusement, l'héritage permet à des objets de fonctionner de la même façon que d'autres.

Le principe de l'héritage

Comme je vous l'ai dit dans l'introduction, la notion d'héritage est l'un des fondements de la programmation orientée objet. Grâce à elle, nous pourrons créer des classes héritées (aussi appelées classes classes dérivées) de nos classes mères (aussi appelées classes classes de base). Nous pourrons créer autant de classes dérivées, par rapport à notre classe de base, que nous le souhaitons. De plus, nous pourrons nous servir d'une classe dérivée comme d'une classe de base pour élaborer encore une autre classe dérivée.

Reprenons l'exemple dont je vous parlais dans l'introduction. Nous allons créer une nouvelle classe, nommée Capitale, héritée de Ville. Vous vous rendrez vite compte que les objets Capitale auront tous les attributs et toutes les méthodes associés aux objets Ville !

```
class Capitale extends Ville {

}
```

C'est le mot clé ***extends*** qui informe Java que la classe Capitale est héritée de Ville. Pour vous le prouver, essayez ce morceau de code dans votre ***main*** :

```
Capitale cap = new Capitale();
System.out.println(cap.decrisToi());
```

Vous devriez avoir la figure suivante en guise de rendu.

Objet Capitale

C'est bien la preuve que notre objet Capitale possède les propriétés de notre objet Ville. Les objets hérités peuvent accéder à toutes les méthodes public (ce n'est pas tout à fait vrai… Nous le verrons avec le mot clé protected) de leur classe mère, dont la méthode decrisToi() dans le cas qui nous occupe.

En fait, lorsque vous déclarez une classe, si vous ne spécifiez pas de constructeur, le compilateur (le programme qui transforme vos codes sources en byte code) créera, au moment de l'interprétation, le constructeur par défaut. En revanche, dès que vous avez créé un constructeur, n'importe lequel, la JVM ne crée plus le constructeur par défaut.

Notre classe Capitale hérite de la classe Ville, par conséquent, le constructeur de notre objet appelle, de façon tacite, le constructeur de la classe mère. C'est pour cela que les variables d'instance ont pu être initialisées ! Par contre, essayez ceci dans votre classe :

```
public class Capitale extends Ville{
  public Capitale(){
    this.nomVille = "toto";
  }
}
```

Vous allez avoir une belle erreur de compilation ! Dans notre classe Capitale, nous ne pouvons pas utiliser directement les attributs de la classe Ville.

Pourquoi cela ? Tout simplement parce les variables de la classe Ville sont déclarées ***private***. C'est ici que le nouveau mot clé ***protected*** fait son entrée. En fait, seules les méthodes et les variables déclarées ***public*** ou ***protected*** peuvent être utilisées dans une classe héritée ; le compilateur rejette votre demande lorsque vous tentez d'accéder à des ressources privées d'une classe mère !

Remplacer ***private*** par ***protected*** dans la déclaration de variables ou de méthodes de la classe Ville aura pour effet de les protéger des utilisateurs de la classe tout en permettant aux objets enfants d'y accéder. Donc, une fois les variables et méthodes privées de la classe mère déclarées en ***protected***, notre objet Capitale aura accès à celles-ci ! Ainsi, voici la déclaration de nos variables dans notre classe Ville revue et corrigée :

```
public class Ville {

  public static int nbreInstances = 0;
  protected static int nbreInstancesBis = 0;
  protected String nomVille;
  protected String nomPays;
  protected int nbreHabitants;
  protected char categorie;

  //Tout le reste est identique.
}
```

Notons un point important avant de continuer. Contrairement au C++, ***Java ne gère pas les héritages multiples :*** *une classe dérivée (aussi appelée classe fille) ne peut hériter que d'une seule classe mère ! Vous n'aurez donc jamais ce genre de classe :*

```
class AgrafeuseBionique extends AgrafeuseAirComprime, AgrafeuseManuelle{

}
```

La raison est toute simple : si nous admettons que nos classes AgrafeuseAirComprime et AgrafeuseManuelle ont toutes les deux une méthode agrafer() et que vous ne redéfinissez pas cette méthode dans l'objet AgrafeuseBionique, la JVM ne saura pas quelle méthode utiliser et, plutôt que de forcer le programmeur à gérer les cas d'erreur, les concepteurs du langage ont préféré interdire l'héritage multiple.

À présent, continuons la construction de notre objet hérité : nous allons agrémenter notre classe Capitale. Comme je vous l'avais dit, ce qui différenciera nos objets Capitale de nos objets Ville sera la présence d'un nouveau champ : le nom d'un monument. Cela implique que nous devons créer un constructeur par défaut et un constructeur d'initialisation pour notre objet Capitale.

Avant de foncer tête baissée, il faut que vous sachiez que nous pouvons faire appel aux variables de la classe mère dans nos constructeurs grâce au mot clé super. Cela aura pour effet de récupérer les éléments de l'objet de base, et de les envoyer à notre objet hérité. Démonstration :

```
class Capitale extends Ville {

  private String monument;

  //Constructeur par défaut
  public Capitale(){
    //Ce mot clé appelle le constructeur de la classe mère
    super();
    monument = "aucun";
  }
}
```

Si vous essayez à nouveau le petit exemple que je vous avais montré un peu plus haut, vous vous apercevrez que le constructeur par défaut fonctionne toujours…

Et pour cause : ici, super() appelle le constructeur par défaut de l'objet Ville dans le constructeur de Capitale. Nous avons ensuite ajouté un monument par défaut.

Cependant, la méthode decrisToi() ne prend pas en compte le nom d'un monument. Eh bien le mot clé super() fonctionne aussi pour les méthodes de classe, ce qui nous donne une méthode decrisToi() un peu différente, car nous allons lui ajouter le champ monument pour notre description :

```
class Capitale extends Ville {
  private String monument;

  public Capitale(){
    //Ce mot clé appelle le constructeur de la classe mère
    super();
    monument = "aucun";
  }
  public String decrisToi(){
    String str =  super.decrisToi() + "\n \t ==>>" + this.monument+ " en est un
monument";
    System.out.println("Invocation de super.decrisToi()");

    return str;
   }
}
```

Si vous relancez les instructions présentes dans le *main* depuis le début, vous obtiendrez quelque chose comme sur la figure suivante.

Utilisation de super

J'ai ajouté les instructions System.out.println afin de bien vous montrer comment les choses se passent.

Bon, d'accord : nous n'avons toujours pas fait le constructeur d'initialisation de Capitale. Eh bien ? Qu'attendons-nous ?

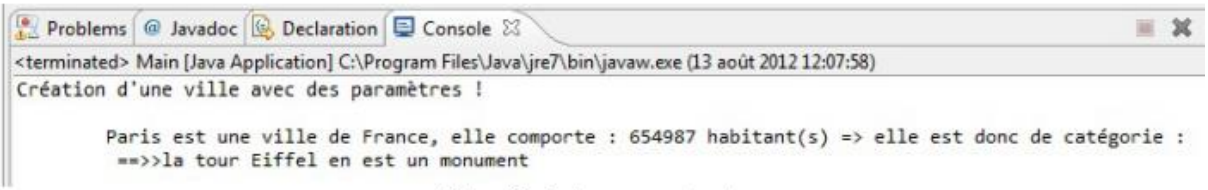
```
public class Capitale extends Ville {

  private String monument;

  //Constructeur par défaut
  public Capitale(){
    //Ce mot clé appelle le constructeur de la classe mère
    super();
    monument = "aucun";
  }

  //Constructeur d'initialisation de capitale
  public Capitale(String nom, int hab, String pays, String monument){
    super(nom, hab, pays);
    this.monument = monument;
  }

  /**
* Description d'une capitale
* @return String retourne la description de l'objet
*/
  public String decrisToi(){
    String str = super.decrisToi() + "\n \t ==>>" + this.monument + "en est un
monument";
    return str;
    }
  /**
* @return le nom du monument
*/
  public String getMonument() {
    return monument;
  }
  //Définit le nom du monument
  public void setMonument(String monument) {
    this.monument = monument;
  }
}
```

Les commentaires que vous pouvez voir sont ce que l'on appelle des commentaires JavaDoc (souvenez-vous, je vous en ai parlé dans le tout premier chapitre de ce cours) : ils permettent de créer une documentation pour votre code. Vous pouvez faire le test avec Eclipse en allant dans le menu Project / Generate JavaDoc.

Dans le constructeur d'initialisation de notre Capitale, vous remarquez la présence de ***super(nom, hab, pays)***;. Cette ligne de code joue le même rôle que celui que nous avons précédemment vu avec le constructeur par défaut. Sauf qu'ici, le constructeur auquel super fait référence prend trois paramètres : ainsi, super doit prendre ces paramètres. Si vous ne lui mettez aucun paramètre, ***super()*** renverra le constructeur par défaut de la classe Ville.

Testez le code ci-dessous, il aura pour résultat la figure suivante.

```
Capitale cap = new Capitale("Paris", 654987, "France", "la tour Eiffel");
  System.out.println("\n"+cap.decrisToi());
```



Classe Capitale avec constructeur

Je vais vous interpeller une fois de plus : vous venez de faire de la méthode decrisToi() une méthode polymorphe, ce qui nous conduit sans détour à ce qui suit.

Le polymorphisme

Voici encore un des concepts fondamentaux de la programmation orientée objet : le polymorphisme.

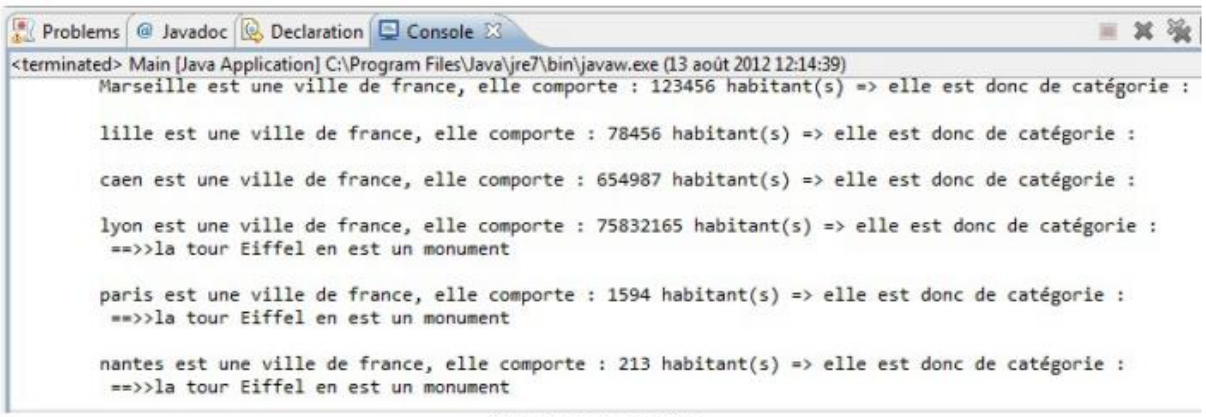
Ce concept complète parfaitement celui de l'héritage, et vous allez voir que le polymorphisme est plus simple qu'il n'y paraît. Pour faire court, nous pouvons le définir en disant qu'il permet de manipuler des objets sans vraiment connaître leur type.

Dans notre exemple, vous avez vu qu'il suffisait d'utiliser la méthode ***decrisToi()*** sur un objet Ville ou sur un objet Capitale. On pourrait construire un tableau d'objets et appeler ***decrisToi()*** sans se soucier de son contenu : villes, capitales, ou les deux.

D'ailleurs, nous allons le faire. Essayez ce code :

```
//Définition d'un tableau de villes null
Ville[] tableau = new Ville[6];

//Définition d'un tableau de noms de villes et un autre de nombres d'habitants
String[] tab = {"Marseille", "lille", "caen", "lyon", "paris", "nantes"};
int[] tab2 = {123456, 78456, 654987, 75832165, 1594, 213};

//Les trois premiers éléments du tableau seront des villes,
//et le reste, des capitales
for(int i = 0; i < 6; i++){
  if (i <3){
    Ville V = new Ville(tab[i], tab2[i], "france");
    tableau[i] = V;
  }

  else{
    Capitale C = new Capitale(tab[i], tab2[i], "france", "la tour Eiffel");
    tableau[i] = C;
  }
}

//Il ne nous reste plus qu'à décrire tout notre tableau !
for(Ville V : tableau){
  System.out.println(V.decrisToi()+"\n");
}
```

La figure suivante vous montre le résultat.



Test de polymorphisme

Nous créons un tableau de villes contenant des villes et des capitales (nous avons le droit de faire ça, car les objets Capitale sont aussi des objets Ville) grâce à notre première boucle for. Dans la seconde, nous affichons la description de ces objets… et vous voyez que la méthode polymorphe ***decrisToi()*** fait bien son travail !

Vous aurez sans doute remarqué que je n'utilise que des objets Ville dans ma boucle : on appelle ceci la ***covariance des variables*** ! Cela signifie qu'une variable objet peut contenir un objet qui hérite du type de cette variable.

Dans notre cas, un objet de type Ville peut contenir un objet de type Capitale. Dans ce cas, on dit que Ville est la superclasse de Capitale.

La covariance est efficace dans le cas où la classe héritant redéfinit certaines méthodes de sa superclasse.

Attention à ne pas confondre la surcharge de méthode avec une méthode polymorphe.

- Une méthode surchargée diffère de la méthode originale par le nombre ou le type des paramètres qu'elle prend en entrée.
- Une méthode polymorphe a un squelette identique à la méthode de base, mais traite les choses différemment. Cette méthode se trouve dans une autre classe et donc, par extension, dans une autre instance de cette autre classe.

Vous devez savoir encore une chose sur l'héritage. Lorsque vous créez une classe (Ville, par exemple), celle-ci hérite, de façon tacite, de la classe Object présente dans Java.

Toutes nos classes héritent donc des méthodes de la classe Object, comme equals() qui prend un objet en paramètre et qui permet de tester l'égalité d'objets. Vous vous en êtes d'ailleurs servis pour tester l'égalité de String() dans la première partie de ce livre.

Donc, en redéfinissant une méthode de la classe Object dans la classe Ville, nous pourrions utiliser la covariance.

La méthode de la classe Object la plus souvent redéfinie est ***toString()*** : elle retourne un String décrivant l'objet en question (comme notre méthode ***decrisToi()***). Nous allons donc copier la procédure de la méthode ***decrisToi()*** dans une nouvelle méthode de la classe Ville : ***toString()***. Voici son code :

```
public String toString(){
  return "\t"+this.nomVille+" est une ville de "+this.nomPays+", elle comporte :
"+this.nbreHabitant+" => elle est donc de catégorie : "+this.categorie;
  }
```

Nous faisons de même dans la classe Capitale :

```
public String toString(){
  String str = super.toString() + "\n \t ==>>" + this.monument + "en est un
monument";
  return str;
  }
```

Maintenant, testez ce code :

```
//Définition d'un tableau de villes null
Ville[] tableau = new Ville[6];

//Définition d'un tableau de noms de villes et un autre de nombres d'habitants
String[] tab = {"Marseille", "lille", "caen", "lyon", "paris", "nantes"};
int[] tab2 = {123456, 78456, 654987, 75832165, 1594, 213};

//Les trois premiers éléments du tableau seront des villes,
//et le reste, des capitales
for(int i = 0; i < 6; i++){
  if (i <3){
    Ville V = new Ville(tab[i], tab2[i], "france");
    tableau[i] = V;
  }

  else{
    Capitale C = new Capitale(tab[i], tab2[i], "france", "la tour Eiffel");
    tableau[i] = C;
  }
}

//Il ne nous reste plus qu'à décrire tout notre tableau !
for(Object obj : tableau){
  System.out.println(obj.toString()+"\n");
}
```

Vous pouvez constater qu'il fait exactement la même chose que le code précédent ; nous n'avons pas à nous soucier du type d'objet pour afficher sa description. Je pense que vous commencez à entrevoir la puissance de Java !

Attention : si vous ne redéfinissez pas ou ne « polymorphez » pas la méthode d'une classe mère dans une classe fille (exemple de toString()), à l'appel de celle-ci avec un objet fille, c'est la méthode de la classe mère qui sera invoquée !

Une précision s'impose : si vous avez un objet v de type Ville, par exemple, que vous n'avez pas redéfini la méthode toString() et que vous testez ce code :

```
System.out.println(v);
```

… vous appellerez automatiquement la méthode toString() de la classe Object ! Mais ici, comme vous avez redéfini la méthode toString() dans votre classe Ville, ces deux instructions sont équivalentes :

```
System.out.println(v.toString());
//Est équivalent à
System.out.println(v);
```

Pour plus de clarté, je conserverai la première syntaxe, mais il est utile de connaître cette alternative.

Pour clarifier un peu tout ça, vous avez accès aux méthodes ***public*** et ***protected*** de la classe Object dès que vous créez une classe objet (grâce à l'héritage tacite). Vous pouvez donc utiliser lesdites méthodes ; mais si vous ne les redéfinissez pas, l'invocation se fera sur la classe mère avec les traitements de la classe mère.

Si vous voulez un exemple concret de ce que je viens de vous dire, vous n'avez qu'à retirer la méthode ***toString()*** dans les classes Ville et Capitale : vous verrez que le code de la méthode main fonctionne toujours, mais que le résultat n'est plus du tout pareil, car à l'appel de la méthode ***toString()***, la JVM va regarder si celle-ci existe dans la classe appelante et, comme elle ne la trouve pas, elle remonte dans la hiérarchie jusqu'à arriver à la classe Object…

Vous devez savoir qu'une méthode n'est « invocable » par un objet que si celui-ci définit ladite méthode.

Ainsi, ce code ne fonctionne pas :

```
//Définition d'un tableau de villes null
Ville[] tableau = new Ville[6];

//Définition d'un tableau de noms de villes et un autre de nombres d'habitants
String[] tab = {"Marseille", "lille", "caen", "lyon", "paris", "nantes"};
int[] tab2 = {123456, 78456, 654987, 75832165, 1594, 213};

//Les trois premiers éléments du tableau seront des villes,
//et le reste, des capitales
for(int i = 0; i < 6; i++){
  if (i <3){
    Ville V = new Ville(tab[i], tab2[i], "france");
    tableau[i] = V;
  }

  else{
    Capitale C = new Capitale(tab[i], tab2[i], "france", "la tour Eiffel");
    tableau[i] = C;
  }
}
```

```
//Il ne nous reste plus qu'à décrire tout notre tableau !
for(Object v : tableau){
  System.out.println(v.decrisToi()+"\n");
}
```

Pour qu'il fonctionne, vous devez dire à la JVM que la référence de type Object est en fait une référence de type Ville, comme ceci :

```
((Ville)v).decrisToi();
```

Vous transtypez la référence v en Ville par cette syntaxe. Ici, l'ordre des opérations s'effectue comme ceci :

- vous transtypez la référence v en Ville ;
- vous appliquez la méthode ***decrisToi()*** à la référence appelante, c'est-à-dire, ici, une référence Object changée en Ville.

Vous voyez donc l'intérêt des méthodes polymorphes : grâce à elles, vous n'avez plus à vous soucier du type de variable appelante. Cependant, n'utilisez le type Object qu'avec parcimonie.

Il y a deux autres méthodes qui sont très souvent redéfinies :

- ***public boolean equals(Object o)***, qui permet de vérifier si un objet est égal à un autre ;
- ***public int hashCode()***, qui attribue un code de hashage à un objet. En gros, elle donne un identifiant à un objet. Notez que cet identifiant sert plus à catégoriser votre objet qu'à l'identifier formellement.

Il faut garder en tête que ce n'est pas parce que deux objets ont un même code de hashage qu'ils sont égaux (en effet, deux objets peuvent avoir la même « catégorie » et être différents…) ; par contre, deux objets égaux ont forcément le même code de hashage ! En fait, la méthode hashcode() est utilisée par certains objets (que nous verrons avec les collections) afin de pouvoir classer les objets entre eux.

La bonne nouvelle, c'est qu'Eclipse vous permet de générer automatiquement ces deux méthodes, via le menu `Source/Generate hashcode and equals`. Voilà à quoi pourraient ressembler ces deux méthodes pour notre objet Ville.

```
public int hashCode() {
  //On définit un multiplication impair, de préférence un nombre
premier
  //Ceci afin de garantir l'unicité du résultat final
  final int prime = 31;
  //On définit un résultat qui sera renvoyé au final
  int result = 1;
  //On ajoute en eux la multiplication des attributs et du
multiplicateur
  result = prime * result + categorie;
  result = prime * result + nbreHabitants;
  //Lorsque vous devez gérer des hashcodes avec des objets dans le
mode de calcul
  //Vous devez vérifier si l'objet n'est pas null, sinon vous aurez
une erreur
  result = prime * result + ((nomPays == null) ? 0 :
nomPays.hashCode());
  result = prime * result + ((nomVille == null) ? 0 :
nomVille.hashCode());
  return result;
}
public boolean equals(Object obj) {
  //On vérifie si les références d'objets sont identiques
  if (this == obj)
    return true;
  //On vérifie si l'objet passé en paramètre est null
  if (obj == null)
    return false;
  //On s'assure que les objets sont du même type, ici de type Ville
  //La méthode getClass retourne un objet Class qui représente la
classe de votre objet
  //Nous verrons ça un peu plus tard...
  if (getClass() != obj.getClass())
    return false;
  //Maintenant, on compare les attributs de nos objets
  Ville other = (Ville) obj;
  if (categorie != other.categorie)
    return false;
  if (nbreHabitants != other.nbreHabitants)
    return false;
  if (nomPays == null) {
    if (other.nomPays != null)
      return false;
  }
  else if (!nomPays.equals(other.nomPays))
    return false;
  if (nomVille == null) {
    if (other.nomVille != null)
      return false;
  }
  else if (!nomVille.equals(other.nomVille))
    return false;

  return true;
}
```

*Il existe encore un type de méthodes dont je ne vous ai pas encore parlé : le type **final**. Une méthode signée **final** est figée, vous ne pourrez jamais la redéfinir (la méthode **getClass()** de la classe Object est un exemple de ce type de méthode : vous ne pourrez pas la redéfinir).*

```
public final int maMethode(){
  //Méthode ne pouvant pas être surchargée
}
```

*Il existe aussi des classes déclarées **final**. Vous avez compris que ces classes sont immuables. Et vous ne pouvez donc pas faire hériter un objet d'une classe déclarée **final** !*

Il en va de même pour les variables déclarées de la sorte.

- Une classe hérite d'une autre classe par le biais du mot clé ***extends***.
- Une classe ne peut hériter que d'une seule classe.
- Si aucun constructeur n'est défini dans une classe fille, la JVM en créera un et appellera automatiquement le constructeur de la classe mère.
- La classe fille hérite de toutes les propriétés et méthodes ***public*** et ***protected*** de la classe mère.
- Les méthodes et les propriétés ***private*** d'une classe mère ne sont pas accessibles dans la classe fille.
- On peut redéfinir une méthode héritée, c'est-à-dire qu'on peut changer tout son code.
- On peut utiliser le comportement d'une classe mère par le biais du mot clé ***super***.
- Grâce à l'héritage et au polymorphisme, nous pouvons utiliser la ***covariance des variables***.
- Si une méthode d'une classe mère n'est pas redéfinie ou « polymorphée », à l'appel de cette méthode par le biais d'un objet enfant, c'est la méthode de la classe mère qui sera utilisée.
- Vous ne pouvez pas hériter d'une classe déclarée ***final***.
- Une méthode déclarée ***final*** n'est pas redéfinissable.