

# La programmation orientée objet en Java

## Partie 2 L'héritage

### CONTENU

le principe de l'héritage .....	1
---------------------------------	---

## LE PRINCIPE DE L'HERITAGE

La notion d'héritage est l'un des fondements de la programmation orientée objet.

Imaginons que, dans le programme réalisé dans la partie 1 (la classe `VilleHeritage`), nous voulions créer un autre type d'objet : des objets `Capitale`. Ceux-ci ne seront rien d'autre que des objets `VilleHeritage` avec un paramètre en plus... disons un monument. Vous n'allez tout de même pas recoder tout le contenu de la classe `VilleHeritage` 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 `VilleHeritage`, 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.

Grâce à cette notion, nous pourrions créer des classes héritées (aussi appelées *classes dérivées*) de nos classes mères (aussi appelées *classes de base*). Nous pourrions créer autant de classes dérivées, par rapport à notre classe de base, que nous le souhaitons. De plus, nous pourrions nous servir d'une classe dérivée comme d'une classe de base pour élaborer encore une autre classe dérivée.

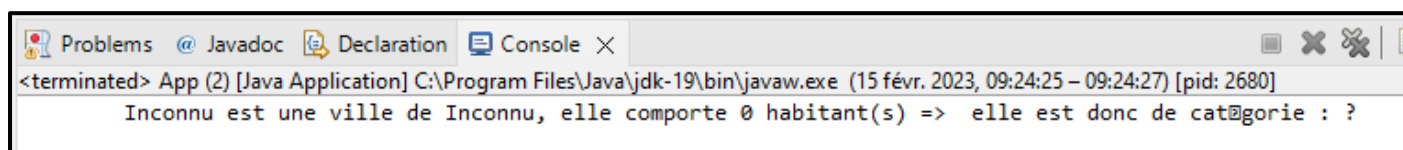
Nous allons créer une nouvelle classe, nommée `Capitale`, héritée de `VilleHeritage`. Vous vous rendrez vite compte que les objets `Capitale` auront tous les attributs et toutes les méthodes associées aux objets `VilleHeritage` !

```
1 package cours_ville;
2
3 public class Capitale extends VilleHeritage {
4
5
6 }
7
```

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

```
1 package cours_ville;
2
3 public class App {
4
5     public static void main(String[] args) {
6
7         Capitale capitale = new Capitale();
8         System.out.println(capitale.decrisToi());
9     }
10 }
```

Vous devriez avoir la figure suivante en guise de rendu :



C'est bien la preuve que notre objet `Capitale` possède les propriétés de notre objet `VilleHeritage`. 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 `VilleHeritage`, 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 ! Cependant, essayez ceci dans votre classe :

```
1 package cours_ville;
2
3 public class Capitale extends VilleHeritage {
4
5     public Capitale()
6     {
7         this.nomVille = "Paris";
8     }
9
10 }
```

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

Pourquoi cela ? Tout simplement parce les variables de la classe `VilleHeritage` 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 `VilleHeritage` 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 `VilleHeritage` revue et corrigée :

```
1 package cours_ville;
2
3 public class VilleHeritage {
4
5     public static int nbInstances = 0;
6     protected static int nbInstancesBis = 0;
7     protected String nomVille;
8     protected String nomPays;
9     protected int nbHabitants;
10    protected char categories;
11 }
```

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 :

```
1 class AgrafeuseBionique extends AgrafeuseAirComprime, AgrafeuseManuelle {
2
3 }
4
```

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 :

```
1 package cours_ville;
2
3 public class Capitale extends VilleHeritage {
4
5     private String monument;
6
7     // Constructeur par défaut
8     public Capitale()
9     {
10         // ce mot clé appelle le constructeur de la classe mère
11         super();
12         monument = "aucun";
13     }
14
15 }
```

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 :

```
public class Capitale extends VilleHeritage {

    private String monument;

    // Constructeur par défaut
    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("utilisation 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 :

```
Creation d'une ville !!!
utilisation de super.decrisToi()
Inconnu est une ville de Inconnu, elle comporte 0 habitant(s) => elle est donc de catégorie : ?
==>> aucun en est un monument
```

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

A présent ajoutons à notre classe `Capitale` un constructeur d'initialisation de `Capitale`, un getters `getMonument()` et un setters `setMonument()`.

```

14 // Constructeur d'initialisation de capitale
15 public Capitale(String pNom, int pHabitant, String pPays, String pMonument) {
16     super(pNom, pHabitant, pPays);
17     this.monument = pMonument;
18 }
19
20 /**
21  * Description d'une capitale
22  * @return String retourne la description de l'objet
23  */
24 public String decrisToi() {
25     String str = super.decrisToi() + "\n\t ==>> "+this.monument+ " en est un monument";
26     System.out.println("utilisation de super.decrisToi()");
27     return str;
28 }
29
30 /**
31  * @return le nom du monument
32  */
33 public String getMonument() {
34     return monument;
35 }
36
37 // Definit le nom du monument
38 public void setMonument(String pMonument) {
39     this.monument = pMonument;
40 }

```

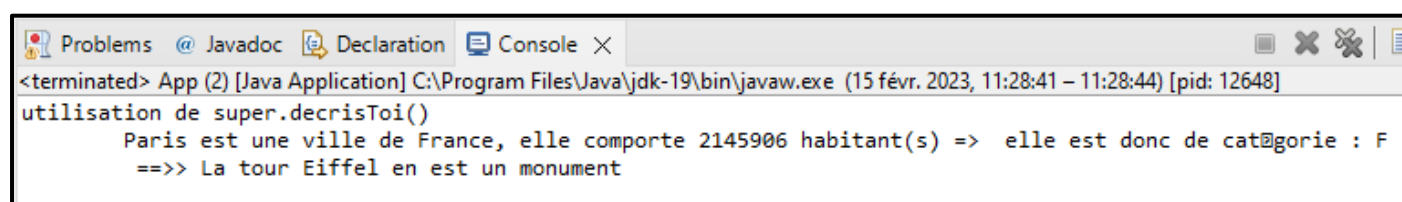
Les commentaires que vous pouvez voir (en bleu clair) sont ce que l'on appelle des commentaires JavaDoc. Ils permettent de créer une documentation pour votre code. Vous pouvez faire le test avec Eclipse en allant dans le menu `Project/Generate JavaDoc`.

Dans le constructeur d'initialisation de notre `Capitale`, vous remarquez la présence de `super(pNom, pHabitant, pPays)`. Cette ligne de code joue le même rôle que celui que nous avons précédemment vu avec le constructeur par défaut. Mais 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.

```

3 public class App {
4
5     public static void main(String[] args) {
6
7         Capitale capitale = new Capitale("Paris", 2145906, "France", "La tour Eiffel");
8         System.out.println(capitale.decrisToi());
9     }
10 }

```



```

<terminated> App (2) [Java Application] C:\Program Files\Java\jdk-19\bin\javaw.exe (15 févr. 2023, 11:28:41 – 11:28:44) [pid: 12648]
utilisation de super.decrisToi()
Paris est une ville de France, elle comporte 2145906 habitant(s) => elle est donc de cat@gorie : F
==>> La tour Eiffel en est un monument

```

```

public static void main(String[] args) {

    VilleHeritage[] tableau = new VilleHeritage[6];

    String tab[] = {"Marseille", "Lyon", "Mulhouse", "Paris", "Washington", "Berlin"};
    int tab1[] = {870321, 522228, 138038, 2945906, 705749, 3748748};
    String tab2[] = {"France", "Etats Unis d'Amerique", "Allemagne"};
    String tab3[] = {"La tour Eiffel", "Le Lincoln Memorial", "le palais du Reichstag"};

    for(int i = 0; i < 6; i++)
    {
        if(i < 3)
        {
            VilleHeritage v = new VilleHeritage(tab[i], tab1[i], "France");
            tableau[i] = v;
        }
        else
        {
            Capitale c = new Capitale(tab[i], tab1[i], tab2[i-3], tab3[i-3]);
            tableau[i] = c;
        }
    }

    for(VilleHeritage V : tableau)
    {
        System.out.println(V.decrisToi()+"\n");
    }
}

```

```

Marseille est une ville de France, elle comporte 870321 habitant(s) => elle est donc de cat@gorie : E

Lyon est une ville de France, elle comporte 522228 habitant(s) => elle est donc de cat@gorie : E

Mulhouse est une ville de France, elle comporte 138038 habitant(s) => elle est donc de cat@gorie : D

utilisation de super.decrisToi()
Paris est une ville de France, elle comporte 2945906 habitant(s) => elle est donc de cat@gorie : F
==>> La tour Eiffel en est un monument

utilisation de super.decrisToi()
Washington est une ville de Etats Unis d'Amerique, elle comporte 705749 habitant(s) => elle est donc de cat@gorie : E
==>> Le Lincoln Memorial en est un monument

utilisation de super.decrisToi()
Berlin est une ville de Allemagne, elle comporte 3748748 habitant(s) => elle est donc de cat@gorie : F
==>> le palais du Reichstag en est un monument

```

--- FIN DU DOCUMENT ---