

Cloner en Java



par Yann D'ISANTO ([Autres articles](#))

Date de publication : 09/10/2006

Dernière mise à jour : 09/10/2006

Voici un petit guide des bonnes pratiques pour cloner vos objets en Java. Il vous montrera comment cloner vos objets en respectant les conventions établies pour ce procédé.

- I - Introduction
- II - L'interface Cloneable
- III - La méthode clone()
- IV - Posséder un attribut non clonable
- V - Hériter d'une classe non clonable
- VI - Remarques
- VII - Liens
- VIII - Remerciements
- IX - Téléchargements

I - Introduction

Dans la programmation orientée objet, il arrive que l'on doive cloner un objet. Le "clonage" d'un objet pourrait se définir comme la création d'une copie par valeur de cet objet. Dans ce tutoriel, nous allons voir comment est implémenté ce concept dans le langage Java.

Remarque : pour bien comprendre ce tutoriel, il est nécessaire de connaître la différence entre un objet immuable et un objet non immuable (cf [Classes et objets immuables](#)).

II - L'interface Cloneable

Pour pouvoir être clonée, une classe doit implémenter l'interface *Cloneable*. Celle-ci indique que la classe peut réécrire la méthode `clone()` héritée de la classe *Object* afin que ses instances puissent procéder à une copie de ses attributs vers une nouvelle instance (copie de l'objet par valeur).

Par convention, les classes implémentant l'interface *Cloneable* doivent réécrire la méthode `Object.clone()` (qui est **protected**) avec une visibilité **public**.

Notez que l'interface *Cloneable* ne contient PAS la méthode `clone()`. Par conséquent, il est impossible de cloner un objet si sa classe ne fait qu'implémenter cette interface. Même si la méthode `clone()` est appelée par réflexion, son succès n'est pas garanti.

Une classe implémentant l'interface *Cloneable* doit nécessairement réécrire la méthode `clone()` pour pouvoir être clonée ; à l'inverse, une classe réécrivant la méthode `clone()` doit implémenter l'interface *Cloneable* sous peine de se retrouver avec une *CloneNotSupportedException*.

III - La méthode clone()

Comme nous l'avons vu, une classe implémentant l'interface *Cloneable* doit réécrire la méthode `clone()`. La méthode `clone()` doit retourner une copie de l'objet que l'on veut cloner. Cette copie dépend de la classe de l'objet, cependant elle doit respecter des conditions (dont certaines par convention).

Pour un objet "clonable" `x` :

l'expression

```
x.clone() != x
```

doit renvoyer `true` ;

l'expression

```
x.clone().getClass() == x.getClass()
```

doit renvoyer `true` (par convention) ;

l'expression

```
x.clone().equals(x)
```

doit renvoyer `true` (par convention).

Par convention, l'objet retourné doit être obtenu par un appel à la méthode `super.clone()` .

Si une classe et toutes ses classes parentes (exceptée la classe *Object*) respectent cette convention, alors l'expression

```
x.clone().getClass() == x.getClass()
```

renvoie bien `true`.

Par convention, l'objet retourné doit être indépendant de l'objet cloné, c'est à dire que tous les attributs non immuables devront être eux aussi clonés.

Il faut aussi connaître l'implémentation par défaut de la méthode `clone()` (méthode `clone()` de la classe *Object*).

La méthode `clone()` de la classe *Object* effectue une opération de clonage spécifique. Dans un premier temps, si la classe de l'objet n'implémente pas l'interface *Cloneable* alors une *CloneNotSupportedException* est levée. Dans le cas contraire, une nouvelle instance de la classe de l'objet est créée puis les attributs sont initialisés avec ceux de l'objet cloné. Cependant la copie de ces attributs se fait par référence et non par valeur (ils ne sont pas clonés !).

On appelle ce procédé une "copie de surface" ("shallow copy") opposée à la "copie en profondeur" ("deep copy") qui, elle, correspond au processus de clonage décrit précédemment.

Donc, pour résumer, les points importants de la réécriture de la méthode `clone()` sont :

- récupérer l'objet à renvoyer en appelant la méthode `super.clone()`,
- cloner les attributs non immuables afin de passer d'une copie de surface à une copie en profondeur de l'objet.

Voici quelques classes illustrant ces propos :

Patronyme.java

```
public class Patronyme implements Cloneable {

    private String prenom;
    private String nom;

    public Patronyme(String prenom, String nom) {
        this.prenom = prenom;
        this.nom = nom;
    }

    public Object clone() {
        Object o = null;
        try {
            // On récupère l'instance à renvoyer par l'appel de la
            // méthode super.clone()
            o = super.clone();
        } catch (CloneNotSupportedException cnse) {
            // Ne devrait jamais arriver car nous implémentons
            // l'interface Cloneable
            cnse.printStackTrace(System.err);
        }
        // on renvoie le clone
        return o;
    }
}
```

Personne.java

```
public class Personne implements Cloneable {

    private Patronyme patronyme;
    private int age;

    public Personne(Patronyme patronyme, int age) {
        this.patronyme = patronyme;
        this.age = age;
    }

    public Object clone() {
        Personne personne = null;
    }
}
```

Personne.java

```
try {
    // On récupère l'instance à renvoyer par l'appel de la
    // méthode super.clone()
    personne = (Personne) super.clone();
} catch(CloneNotSupportedException cnse) {
    // Ne devrait jamais arriver car nous implémentons
    // l'interface Cloneable
    cnse.printStackTrace(System.err);
}

// On clone l'attribut de type Patronyme qui n'est pas immuable.
personne.patronyme = (Patronyme) patronyme.clone();

// on renvoie le clone
return personne;
}
```

Jouet.java

```
public class Jouet implements Cloneable {

    private String nom;

    public Jouet(String nom) {
        this.nom = nom;
    }

    public Object clone() {
        Object o = null;
        try {
            // On récupère l'instance à renvoyer par l'appel de la
            // méthode super.clone()
            o = super.clone();
        } catch(CloneNotSupportedException cnse) {
            // Ne devrait jamais arriver car nous implémentons
            // l'interface Cloneable
            cnse.printStackTrace(System.err);
        }
        // on renvoie le clone
        return o;
    }
}
```

Enfant.java

```
public class Enfant extends Personne {

    private Jouet jouetPrefere;

    public Enfant(Patronyme patronyme, int age, Jouet jouetPrefere) {
        super(patronyme, age);
        this.jouetPrefere = jouetPrefere;
    }

    public Object clone() {
        // On récupère l'instance à renvoyer par l'appel de la
        // méthode super.clone() (ici : Personne.clone())
        Enfant enfant = (Enfant) super.clone();

        // On clone l'attribut de type Jouet qui n'est pas immuable.
        enfant.jouetPrefere = (Jouet) jouetPrefere.clone();

        // on renvoie le clone
        return enfant;
    }
}
```

Enfant.java

}

Voici un code pour mettre en pratique ces classes :

CloneMain.java

```
public class CloneMain {  
  
    public static void main(String []args) {  
        Personne personnel = new Personne(new Patronyme("Jean", "Dupond"), 30);  
        Personne personne2 = (Personne) personnel.clone();  
        System.out.println(personnel);  
        System.out.println(personne2);  
        Enfant enfant1 = new Enfant(new Patronyme("Pierre", "Dupond"), 10, new Jouet("Teddy bear"));  
        Enfant enfant2 = (Enfant) enfant1.clone();  
        System.out.println(enfant1);  
        System.out.println(enfant2);  
    }  
}
```

Ce qui donne la sortie suivante :

```
Personne@923e30  
Personne@130c19b  
Enfant@11b86e7  
Enfant@35ce36
```

IV - Posséder un attribut non clonable

Dans le cas où votre classe possède un attribut non immuable et non clonable, vous ne pouvez tout simplement pas le cloner. Bien que vous puissiez utiliser un constructeur par copie de votre attribut (s'il en possède un), vous pouvez vous retrouver dans le cas où vos deux objet (original et clone) possède le même attribut (même instance).

Notez que cette situation n'est pas "anormale" (ne paniquez pas si elle se présente à vous), il est en effet possible que l'on veuille délibérément garder la même instance d'un attribut.

Après tout, ce n'est peut être pas pour rien qu'il n'implémente pas l'interface *Cloneable*.

V - Hériter d'une classe non clonable

Dans le cas où votre classe hérite d'une classe non clonable, sachez que l'appel à la méthode `super.clone()` revient à exécuter la méthode `clone()` par défaut (copie de surface). Cela veut dire que les attributs non immuables de la classe parente ne seront pas clonés !

C'est à vous qu'il revient de les cloner, à moins qu'ils soient inaccessibles (déclarés **private**) ou non clonables.

VI - Remarques

Précisons que la classe `Object` n'implémente pas l'interface `Cloneable`, donc appeler la méthode `clone()` sur un objet dont la classe est `Object` lèvera invariablement une `CloneNotSupportedException`.

Notons également que les tableaux sont considérés comme implémentant l'interface `Cloneable`. Cependant c'est la méthode `clone()` par défaut qui est utilisée (copie de surface), les objets contenus dans les deux tableaux sont donc copiés par référence.

CloneTableaux.java

```
public class CloneTableaux {

    public static void main(String []args) {
        Personne personne1 = new Personne(new Patronyme("Pierre", "Dupond"), 31);
        Personne personne2 = new Personne(new Patronyme("Paul", "Dupond"), 32);
        Personne[] array1 = { personne1, personne2 };
        Personne[] array2 = (Personne[]) array1.clone();
        System.out.println("array1 : " + array1);
        System.out.println("array2 : " + array2);
        System.out.println("array1[0] : " + array1[0]);
        System.out.println("array2[0] : " + array2[0]);
    }
}
```

La sortie générée doit ressembler à ceci :

```
array1 : [LPersonne;@923e30
array2 : [LPersonne;@130c19b
array1[0] : Personne@1f6a7b9
array2[0] : Personne@1f6a7b9
```

On voit bien que les deux tableaux sont deux instances différentes, mais on remarque aussi que les objets contenus pointent vers la même instance, ils n'ont donc pas été clonés.

Une dernière remarque, depuis Java 5 il est possible de changer le type de retour d'une méthode que l'on réécrit à condition que celui-ci dérive du type original. Dans notre cas cette fonctionnalité se révèle particulièrement intéressante car cela nous permet d'éviter les transtypages lors de l'appel à la méthode `clone()`.

Voici les nouvelles méthodes `clone()` de nos classes :

```
// Méthode clone() de la classe Patronyme
// Renvoie maintenant un objet de type Patronyme et non plus de type Object
public Patronyme clone() {
    Patronyme patronyme = null;
    try {
        patronyme = (Patronyme) super.clone();
    } catch (CloneNotSupportedException cnse) {
        cnse.printStackTrace(System.err);
    }
}
```

```
    }
    return patronyme;
}

// Méthode clone() de la classe Personne
// Renvoie maintenant un objet de type Personne et non plus de type Object
public Personne clone() {
    Personne personne = null;
    try {
        personne = (Personne) super.clone();
    } catch(CloneNotSupportedException cnse) {
        cnse.printStackTrace(System.err);
    }
    // Plus besoin de transtyper pour cloner le patronyme.
    personne.patronyme = patronyme.clone();
    return personne;
}

// Méthode clone() de la classe Jouet
// Renvoie maintenant un objet de type Jouet et non plus de type Object
public Jouet clone() {
    Jouet jouet = null;
    try {
        jouet = (Jouet) super.clone();
    } catch(CloneNotSupportedException cnse) {
        cnse.printStackTrace(System.err);
    }
    return jouet;
}

// Méthode clone() de la classe Enfant
// Renvoie maintenant un objet de type Enfant et non plus de type Object
public Enfant clone() {
    Enfant enfant = (Enfant) super.clone();
    // Plus besoin de transtyper pour cloner le jouet.
    enfant.jouetPrefere = jouetPrefere.clone();
    return enfant;
}
```

et voici le nouveau code pour cloner nos objets :

```
public static void main(String []args) {
    Personne personnel = new Personne(new Patronyme("Jean", "Dupond"), 30);
    // Plus besoin de transtyper pour cloner la personne.
    Personne personne2 = personnel.clone();
    System.out.println(personnel);
    System.out.println(personne2);
    Enfant enfant1 = new Enfant(new Patronyme("Pierre", "Dupond"), 10, new Jouet("Teddy bear"));
    // Plus besoin de transtyper pour cloner l'enfant.
    Enfant enfant2 = enfant1.clone();
    System.out.println(enfant1);
    System.out.println(enfant2);
}
```

VII - Liens

- **La Javadoc**
 - **L'interface Cloneable**
 - **La méthode Object.clone()**
- **Classes et objets immuables**

VIII - Remerciements

Je remercie **Elmilouse**, **@om** et **Swoög** pour leurs remarques et corrections.

IX - Téléchargements

Article au format pdf : [FTP \(lien principal\)](#), [HTTP \(lien de secours\)](#)

Article au format html : [FTP \(lien principal\)](#), [HTTP \(lien de secours\)](#)

Code source des exemples de l'article : [FTP \(lien principal\)](#), [HTTP \(lien de secours\)](#)

