La méthode clone()

Philippe PRADOS

pp@philippe.prados.name

Préservez l'environnement,

n'imprimez pas ce document

TABLE DES MATIERES

1. Object.clone() ... 3
2. Constructeur de copie ... 4
3. Instance immuable ... 4
4. Une interface Clone ... 5
5. La méthode copy ... 6
6. Conclusion ... 7

Avant-propos

Comment rédiger la méthode clone() ? Quand utiliser Cloneable ? Ce document répond à ces questions.

Il est souvent nécéssaire de dupliquer une instance. Cela permet de gérer correctement les exceptions et les agrégations. Nous désirons proposer une méthode `public clone()` retournant une copie d'une instance.

```
public class Personne
{ public Object clone()
  { ...
  }
}
```

Il y a plusieurs approches possibles.

1. OBJECT.CLONE()

La classe `Object` de java propose une méthode `protected clone()` construisant automatiquement une instance similaire à `this`. Tous les attributs de cette nouvelle instance sont initialisés avec les attributs de `this`. Pour bénéficier de cela, il faut signaler à la méthode `Object.clone()` qu'une classe autorise la duplication d'une instance par ce mécanisme. Pour cela, la classe doit implémenter l'interface `Cloneable`. Cette interface ne propose pas de service. La méthode `Object.clone()` vérifie que l'instance courante implémente cette interface. Dans le cas contraire, la méthode génère une exception.

```
public class Object
{
  protected Object clone() throws CloneNotSupportedException
  { if (!(this instanceof Cloneable))
      throw new CloneNotSupportedException();
    ...
  }
}
```

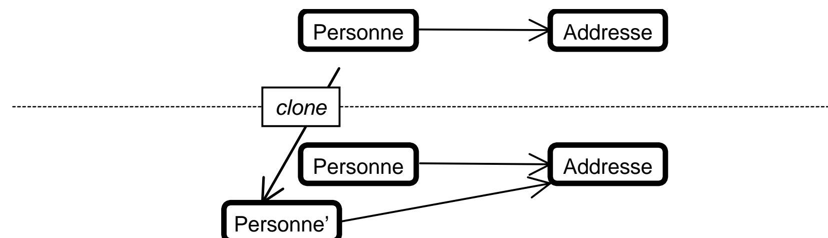
Pour bénéficier de cette méthode, il faut la surcharger et la rendre publique.

```
public class Personne implements Cloneable
{
  public Object clone() throws CloneNotSupportedException
  { return super.clone();
  }
}
```

Etant sure que l'exception `CloneNotSupportedException` ne sera jamais emise, il est préférable de la supprimer de la signature.

```
public class Personne implements Cloneable
{
  public Object clone()
  { try
    { return super.clone();
    } catch (CloneNotSupportedException x)
    { return InternalError("N'arrive jamais");
    }
  }
}
```

Il faut se méfier de cette méthode. En effet, elle considère que tous les attributs objets sont des relations. Ce n'est généralement pas le cas. Par exemple, si une `Personne` agrège une `Addresse`, la méthode `Object.clone()` retournera une nouvelle instance utilisant la même `Addresse` que `this`.



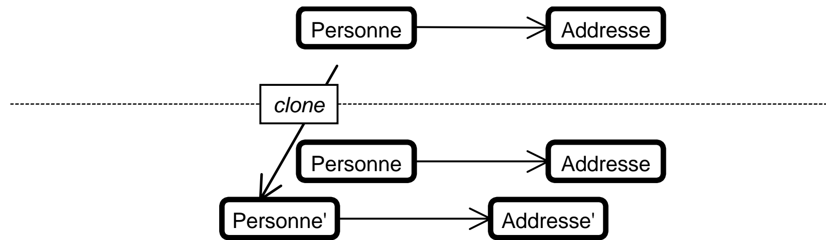
Il faut enrichir la méthode `clone()` pour tenir compte des agrégations.

```
public class Personne implements Cloneable
{ Addresse _adresse;

  public Object clone()
  { try
    { Personne obj=(MaClass)super.clone();
      obj._addresse=(Addresse)_address.clone(); // Clone l'agrégat
      return obj;
```

```
    } catch (CloneNotSupportedException x)
    { return InternalError("N'arrive jamais");
    }
  }
}
```

Cela permet de dupliquer les agrégations.



La méthode `Object.clone()` permet d'avoir une instance sans appeler de constructeur. C'est pour cela que cette méthode est `native`. Ce point est très important. S'il doit y avoir des traitements particuliers dans un constructeur ayant des effets de bord en dehors de l'instance courante, il ne faut pas utiliser cette méthode. En effet, il y a un risque de construire une instance sans appeler le constructeur. Il est alors préférable d'utiliser un constructeur par défaut ou d'offrir un constructeur de copie.

Pour être compatible avec le multitâche, la méthode `clone()` doit être déclarée `synchronized`.

Une classe critique vis-à-vis de la sécurité ne doit pas implémenter l'interface `Cloneable` afin d'éviter des utilisations frauduleuses.

2. Constructeur de copie

Un constructeur de copie est un constructeur recevant un paramètre du même type que la classe.

```
public class Personne
{
  public Personne(Personne x) // Constructeur de copie
  { //...
  }
}
```

Cela permet de construire une instance à partir d'une autre. Cette approche n'est pas polymorphique, contrairement à l'utilisation de la méthode `clone()`. Si une classe propose cette méthode, il est possible de rédiger la méthode `clone()` avec l'aide du constructeur de copie.

```
public class Personne
{
  public Personne(Personne x) // Constructeur de copie
  { //...
  }
  public Object clone()
  { return new Personne(this);
  }
}
```

Le constructeur de copie ne peut pas bénéficier de la méthode `Object.clone()` car il doit modifier l'instance courante et non créer une nouvelle instance. Dans cette situation, il n'est pas nécessaire de déclarer que la classe implémente l'interface `Cloneable`. La méthode `Object.clone()` n'étant pas appelée, cette interface n'est pas nécessaire.

Le constructeur de copie n'apporte pas grand-chose par rapport à la méthode `clone()`.

```
Personne p=new Personne(i);
```

Il permet d'éviter de vérifier si l'instance n'est pas à `null`, mais au détriment du polymorphisme.

```
Personne p=(i==null) ? null : i.clone();
```

Par contre, il est souvent utile à la rédaction de la méthode `clone()`.

3. Instance immuable

Les instances immuables sont des instances qui ne modifient pas leurs états lors de leurs durées de vie. La classe `String` est une instance immuable. Une fois qu'une instance `String` est construite, il n'est plus possible de la modifier. La méthode `clone()` d'une instance immuable peut retourner `this`.

```
public class Immuable
{
  public Object clone()
  { return this;
  }
}
```

Les instances immuables permettent de partager des instances, mais obligent à créer de nouveaux objets lors de toutes manipulations. Par exemple, la méthode `toLowerCase()` de la classe `String` retourne une nouvelle instance.

4. Une interface Clone

Pour pouvoir agréger une instance, il est souvent nécessaire de pouvoir la dupliquer. Il faut proposer une interface permettant de normaliser l'appel à la méthode `clone()`.

```
public interface Clone
{ public Object clone();
}
```

Ainsi, les instances proposant cette méthode publique, peuvent implémenter cette interface.

```
public class Personne implements Clone
{
  public Object clone()
  { ...
  }
}
```

Une instance `Personne` peut être agrégé. On peut utiliser une instance `Personne` avec une sémantique par valeur. Par exemple, pour recevoir la valeur d'une instance `Personne`, une méthode peut cloner son paramètre.

```
public void f(Personne p)
{ Personne value=p.clone();
  ...
}
```

Cela permet d'avoir une nouvelle instance n'ayant pas d'effet de bord sur le paramètre. La méthode `f()` ne modifie pas l'instance pointée par `p`. Cela est équivalent à l'utilisation d'un type primitif.

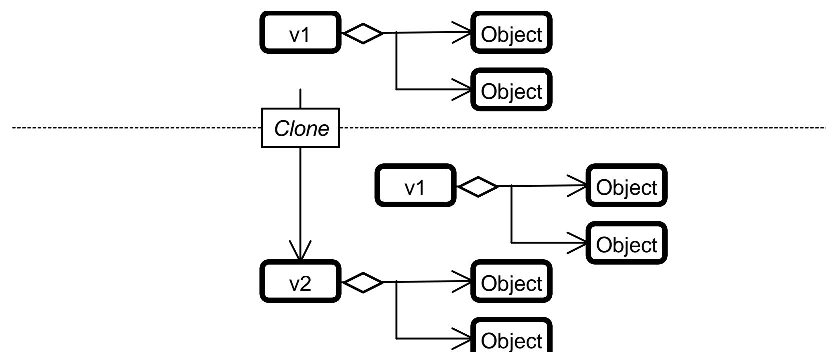
```
public void f(int p)
{ ...
}
```

Avec l'interface `Clone`, il est possible de rédiger de nouveaux conteneurs de type `Vector`, mais fonctionnant sur le principe de l'agrégation. Un `VectorAgr` agrège ses éléments.

```
class VectorAgr implements Clone
{ private Vector _vec=new Vector();
  public VectorAgr()
  { }
  public VectorAgr(VectorAgr obj)
  { for (Enumeration e=obj._vec.elements();e.hasMoreElements();)
    { Clone elem=(Clone)e.nextElement();
      _vec.addElement(elem.clone());
    }
  }
  public synchronized Object clone()
  { return new VectorAgr(this);
  }
  public synchronized void addElement(Clone obj)
  { _vec.addElement(obj);
  }
  //...
}
```

Le clone d'une instance de ce type clonera les instances du vecteur.

```
VectorAgr v1=new VectorAgr();
...
VectorAgr v2=(VectorAgr)v1.clone();
```



L'interface Clone normalise l'utilisation de la méthode clone(). Cela permet d'utiliser les objets avec une sémantique par valeur.

5. LA METHODE COPY

Nous avons vu l'interaction entre la méthode clone() et le constructeur de copie. Cela permet de manipuler la valeur d'une instance. Pour copier la valeur d'une instance dans une autre instance, il faut proposer une méthode copy(). Cela est l'équivalent de l'opérateur égal des types primitifs.

```
{ int a,b;
  a=b;
}
{ Personne a=new Personne(),b=new Personne();
  a.copy(b); // a=b;
}
```

La méthode copy() est importante pour les performances d'un programme Java. En effet, elle permet de recycler une instance. Cela réduit le travail du ramasse miettes.

Pour rédiger la méthode copy(), il faut dans un premier temps libérer les ressources utilisées par l'instance courante (appel de finalize()), puis utiliser le constructeur de copie. Cela n'étant pas autorisé par le langage, il faut séparer le constructeur de copie en deux.

```
class Personne
{ protected final void cctr(MaClass x)
  { // Copie les attributs
    ...
  }
  // Constructeur de copie
  public Personne(Personne x)
  { cctr(x);
  }
  public Object clone()
  { return new Personne(this);
  }
  public final void copy(Personne x)
  { finalize();
    cctr(x);
  }
  public void finalize()
  { // Libère les ressources
    ...
  }
}
```

cctr() doit être déclaré final car une méthode polymorphique ne doit pas être appelée par un constructeur. Cela permet également au compilateur d'optimiser le code en dupliquant le corps de la méthode cctr() dans le constructeur de copie et dans la méthode copy().

Cela n'est malheureusement pas suffisant. Comment rédiger une sous-classe de Personne ?

```
class Sportif extends Personne
{ protected final void cctr(Sportif x)
  { super.cctr((Personne)x);
    // Copie les attributs
    ...
  }
  // Constructeur de copie
  public Sportif(Sportif x)
  { super(...
    cctr(x);
  }
  public Object clone()
  { return new Sportif(this);
  }
}
```

Le constructeur de copie de la sous-classe, doit appeler un constructeur hérité. S'il appelle le constructeur de copie de Personne, cela a pour conséquence d'appeler la méthode Personne.cctr() deux fois. Ce n'est pas souhaitable.

```
public Sportif(Sportif x)
{ super(this); // Premier appel à Personne.cctr()
  cctr(x);     // Deuxieme appel à Personne.cctr()
}
```

Déclarer la méthode cctr() non final n'est pas la solution. En effet, cette méthode sera alors appelée par le constructeur de Personne alors que les attributs spécifiques à Sportif ne sont pas encore initialisés.

Il est possible d'appeler le constructeur par défaut s'il existe, et s'il n'a pas d'effet de bord indésirable.

```
public Sportif(Sportif x)
{ super(); // Constructeur par defaut
```

```
  cctr(x);
}
```

Autrement, il est nécessaire de proposer un constructeur technique particulier n'ayant pas d'effet de bord. Pour sélectionner ce constructeur, il faut utiliser un type particulier indiquant de ne pas initialiser l'instance.

```
class Personne
{ protected interface WithoutInit  { }
  protected Personne(WithoutInit x) { }
  ...
}

class Sportif extends Personne
{ protected Sportif(WithoutInit x)
  { super(x);
  }
  public Sportif(Sportif x)
  { super((WithoutInit)null); // Constructeur sans init
    cctr(x);
  }
  public Object clone()
  { return new Sportif(this);
  }
  ...
}
```

La méthode `copy()` doit être réécrite dans toutes les sous-classes en modifiant le paramètre d'entrée.

```
class Sportif extends Personne
{ ...
  public final void copy(Sportif x)
  { finalize();
    cctr(x);
  }
}
```

On peut dans ce cas s'abstenir de proposer le constructeur de copie.

```
class Sportif extends Personne
{ protected Sportif(WithoutInit x)
  { super(x);
  }
  public Object clone()
  { Sportif s=new Sportif((WithoutInit)null);
    s.cctr(this);
    return s;
  }
  ...
}
```

6. Conclusion

La méthode `clone()`étant proposé dans la classe `Object`, il n'est pas possible de la surcharger avec un type de retour différent de `Object`. Cela oblige, même si on n'utilise pas la méthode `Object.clone()`, à retourner le type `Object`. L'utilisateur de la méthode est généralement contraint de convertir le code retour.

```
Personne p=new Personne();
Personne p2=(Personne)p.clone();
```

Si la méthode `clone()` n'était pas présente dans la classe `Object`, et serait proposée par exemple comme une fonction statique de la classe `System`, il n'y aurait pas ce problème. Il ne faut pas oublier que cette méthode est utilisable que pour les instances proposant l'interface `Cloneable`. Ce n'est pas un service par défaut pour tous les objets.

L'étonnant dans l'implémentation de la méthode `Object.clone()`, est qu'il y a deux approches de la protection. Cette méthode est `protected`, et ne peut être appelé qu'avec des instances implémentant l'interface `Cloneable`. Sinon, cela génère une exception. Il aurait été préférable d'offrir une méthode statique dans la classe `System`.

```
public class System
{ ...
  public static native Object clone(Cloneable x);
}
```

Ainsi, cette méthode ne peut être appelée que par les instances implémentant l'interface `Cloneable`. Cela est vérifié par le compilateur. Il n'est pas nécessaire de générer une exception. Si les concepteurs de Java avaient fait ce choix, il aurait été possible de proposer une méthode `clone()` retournant un type différent de `Object`.

D'autre part, si Java permettait de surcharger une méthode avec un code retour différent, mais respectant la hiérarchie, le code serait simplifié. Le C++ propose cela. Il serait alors possible d'avoir :

```
class Personne
{
  public Personne(Personne x) // Constructeur de copie
  { //...
  }
  public Personne clone()
  { return new Personne(this);
  }
}

...
Personne p=new Personne();
Personne p2=p.clone();
```

Les conversions ne deviennent plus nécessaire car générées automatiquement par le compilateur. Une prochaine version de Java offrira peut-être cette extension. Il ne s'agit que d'une extension du compilateur, pas de la machine virtuelle. Le compilateur est capable de générer automatiquement une conversion lors de l'appel d'une méthode de ce type.