# Les attributs

## Présentation

Un attribut est, conceptuellement, une association unidirectionnelle en direction de l'attribut. L'attribut n'a donc aucune responsabilité dans la relation. Pratiquement, chaque objet gardera une copie qui lui est propre de chacun de ses attributs.

Il faut **empêcher** un objet extérieur de **modifier** directement la valeur d'un attribut. Pour modifier un tel attribut, il doit être obligatoire de le faire via l'objet possédant l'attribut.

Par exemple, on enregistre dans un client sa commande. Pour modifier la commande, il faudrait le demander à l'objet qui la possède, à savoir le client. Les extraits de code suivants illustrent cela :

```
public class Client {
            private Commande commande;
            public void enregistrer(Commande commande) {
                  this.commande = commande;
            }
            ...
}
public class Main {
      public static void main(String[] args) {
            Client client = new Client();
            Commande com = new Commande();

            client.enregistrer(com);

            client.acheterArticle() ;
```

// Mais on détient encore **la référence de la commande du client** à laquelle on peut ajouter ou supprimer ce que l'on veut… La ligne de code qui suit ne devrait **pas impacter** la commande du client

```
            com.ajouterArticle() ;
                        //→ ne devrait rien changer dans la commande du client
            }
   }
```

Un objet doit rester **garant des attributs** qu'il renferme. C'est pour cela que les modifications de ceux-ci doivent passer par lui car il en est **responsable**. Il faut gérer cela de façon programmatique afin que cette responsabilité soit respectée.

Remarquons que certains types java sont **immuables** ; il s'agit des types primitifs (tels que `int, long, boolean, …`) ou encore les classes Java telles que `String` ou `LocalDateTime`) Ceci est indiqué dans la java API en *Implementation Requirements: This class is immutable and thread-safe*.

Les attributs de ce type ne posent pas ce problème : ils ne sont pas modifiables.

## Les attributs unaires

Les attributs unaires sont des attributs représentés par un champ qui n'est pas une collection. On veillera à cloner les accès à de tels attributs.

Reprenons notre exemple avec la commande et modifions-le afin de résoudre le problème indiqué :

```
public class Client {
            private Commande commande;
            public void enregistrer(Commande commande) {
                  this.commande = commande.clone();
            }
            …
}
```

Si la classe représentant l'attribut est écrite par vos soins, il faut la rendre **`Cloneable`** et écrire la méthode `clone()`.

```
public class Commande implements Cloneable {
         public Commande clone() {
           try {
             return (Commande)super.clone();
           }catch(CloneNotSupportedException e) {
               throw new InternalError();
           }
         }
```

L'appel à `super.clone()` fait un **clone superficiel**. Si l'objet référence d'autres objets, les **références** de ceux-ci sont simplement **recopiées** bit à bit ; ces objets ne sont pas clonés. Si on désire qu'ils le soient, il faut le faire explicitement.

Dans notre exemple, si client doit également être *clonable* :

```
      public Client clone() {
            try {
              Client c = (Client)super.clone();
              c.commande = (Commande)c.commande.clone();
              return c;
            }catch (CloneNotSupportedException e) {
                  throw new InternalError();
            }
      }
```

## Les attributs multiples

Si un attribut est une Collection, la cloner ne sert à rien, car le clone n'est que superficiel. Si on désire renvoyer une collection formée de clones, il faut en créer une nouvelle et la remplir de clones ; on appelle cela un **clonage en profondeur**.

Supposons que la commande possède des lignes de commandes :

```
   public List<LigneDeCommande> lignesDeCommandes() {
            List<LigneDeCommande> l = new ArrayList<LigneDeCommande>();
            for (LigneDeCommande lc: lignes)
               try {
                  l.add((LigneDeCommande)lc.clone());
              }catch(CloneNotSupportedException e) {
                  throw new InternalError();
              }
            return l;
      }
```

Si on désire renvoyer un `Iterator`, il faut en définir un en classe interne :

```
   public Iterator<LigneDeCommande> lignes() {
            return new ListIteraor();
      }
   private class ListIterator implements Iterator<LigneDeCommande> {
      private Iterator<LigneDeCommande> it = lignes.iterator();
      public boolean hasNext() {
              return it.hasNext();
      }
      public LigneDeCommande next() {
             try {
                  return it.next().clone();
              }catch(CloneNotSupportedException e) {
                  throw new InternalError();
              }
      }
      public void remove() {
         throw new UnsupportedOperationException();
    }
}
```