

La méthode equals()

Philippe PRADOS

pp@philippe.prados.name



*Préservez l'environnement,
n'imprimez pas ce document*

TABLE DES MATIERES

1.	Approche rigide	4
2.	Approche souple.....	5
3.	Offrir les deux approches.....	5
4.	Approche rigide étendue.....	6
5.	Approche souple étendue.....	7
6.	Le lien avec la méthode <code>hashCode()</code>	7
7.	Pourquoi est-ce si complexe ?	8
8.	Comment comparer les attributs ?	8
8.1	Les attributs primitifs	8
8.2	Les relations	8
8.3	Les agrégations	9
8.4	Les caches	10
8.5	Multitâche.....	10
8.6	Optimisation.....	10
8.6.1	Comparer avec <code>this</code>	10
8.6.2	Trier les attributs.....	11
8.6.3	Les instances immuables.....	11
8.6.4	Mémoriser la valeur de hash.....	11
9.	Conclusion	12

Avant-propos

La méthode `equals()` de Java est très difficile à rédiger. Ce document explique pourquoi et les différentes démarches possibles.

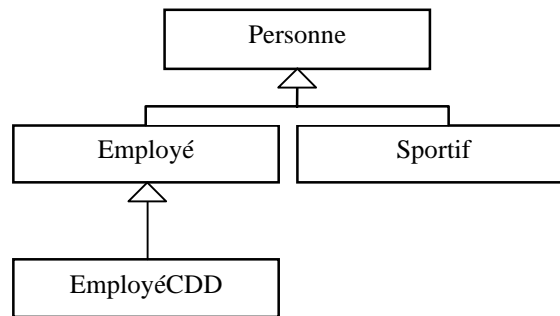
La classe `Object` de Java propose la méthode `equals()`. Cette méthode permet de comparer la valeur de deux instances. Par défaut, l'implémentation retourne `true` si le paramètre est égal à `this`. Une instance a toujours la même valeur qu'elle-même. Par contre, deux instances différentes peuvent avoir également la même valeur. Par exemple, deux variables de type `int`, peuvent avoir la valeur 3. Deux instances de la classe `Adresse` peuvent avoir les mêmes valeurs. La méthode `equals()` est là pour cela.

```
Adresse a1=new Adresse("Chez moi");
Adresse a2=new Adresse("Chez moi");
a1==a2;           // false
a1.equals(a2);   // true
```

Il est théoriquement possible de comparer toutes instances avec n'importe quelle autre. Il doit être possible de comparer une `Adresse` avec une `Date`. Une `Adresse` n'étant pas une `Date`, la méthode doit retourner `false`. Cela se complique s'il existe un arbre d'héritage.

Il faut que la comparaison fonctionne malgré l'héritage. A titre d'illustration, nous allons choisir le modèle objet suivant :

- Une classe `Personne`,
- Une classe `Employé` qui hérite de `Personne`,
- Une classe `EmployéCDD` qui hérite d'`Employé`,
- Une classe `Sportif` qui hérite de `Personne`.



Un `Employé` est une sorte de `Personne`. Tous les traitements possibles avec une `Personne` doivent être valides avec un `Employé`. Il est possible de comparer si deux instances `Personne` ont les mêmes valeurs, il doit donc être possible de comparer une `Personne` est un `Employé`. Si une `Personne` possède les mêmes valeurs que l'employé au niveau des attributs de `Personne` (nom, prénom,...), la méthode `equals()` peut retourner `true`.

```
Personne p=new Personne("Bill","Porte");
Employé e=new Employé("Bill","Porte");
p.equals(e); // true
```

En fait, il existe deux stratégies de comparaison.

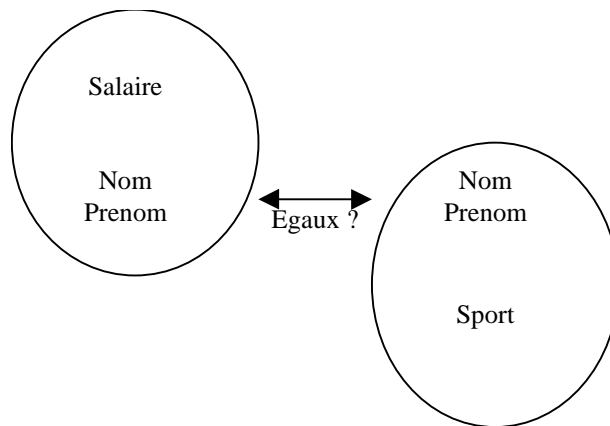
- Comparer deux instances c'est comparer l'union des attributs de chaque instance. Si l'une d'elles ne possède pas un des attributs à comparer, l'égalité n'est pas résolue. Cette approche est appelée : rigide.
- Ou bien, comparer deux instances de types différents peut consister à comparer uniquement l'intersection des attributs de chaque instance. Cette approche est appelée : souple.

Par exemple, la classe `Personne` possède les attributs `nom_` et `prenom_`. La classe `Employé` possède un attribut supplémentaire, le `salaire_`. Un `Sportif` possède un attribut `sport_`. Si on désire comparer un `Employé` et un `Sportif`, il y a deux approches possibles.

- Approche rigide : $\text{Employé} \cup \text{Sportif} = [\text{nom_}, \text{prenom_}, \text{salaires_}, \text{sport_}]$. Si l'un de ces attributs n'est pas présent dans l'une des deux instances, la méthode `equals()` retourne `false`.
- Approche souple : $\text{Employé} \cap \text{Sportif} = [\text{nom_}, \text{prenom_}]$. Si ces deux attributs sont identiques, la méthode `equals()` retourne `true`.

Il faut être capable de proposer l'approche rigide ou souple pour la méthode `equals()`.

1. APPROCHE RIGIDE



La méthode `equals()` version rigide peut être rédigée ainsi :

```
public class Personne
{ //...
  // Version rigide
  public boolean equals(Object x)
  { if ((x!=null) && (x.getClass()==Personne.class))
    { // Comparer les attributs
      return (...
    }
    return false;
  }
}
```

Dans ce cas, il n'est pas possible de bénéficier de la méthode `Personne.equals()` dans une sous-classe. Il est préférable de découper la méthode en deux pour différencier le test des attributs et la vérification de la classe.

```
public class Personne
{ //...
  // Version rigide
  protected final boolean equals_(Personne obj)
  { // Comparer les attributs
    return (...
  }

  public boolean equals(Object x)
  { if ((x!=null) && (x.getClass()==Personne.class))
    { return equals_((Personne)x);
    }
    return false;
  }
}
```

La méthode `equals()` version rigide de la classe `Employé` peut alors être rédigée ainsi :

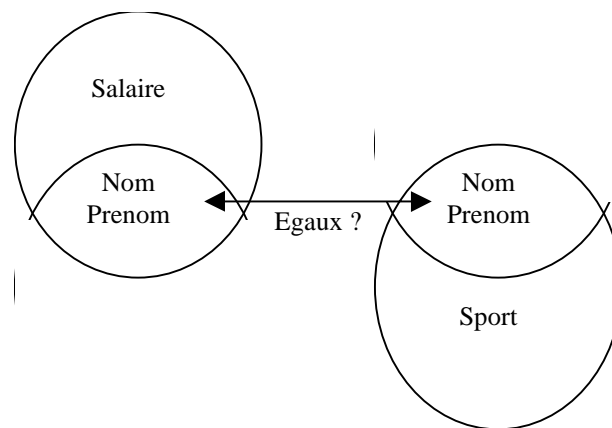
```
public class Employé extends Personne
{ //...
  // Version rigide
  protected final boolean equals_(Employé obj)
  { // Comparer les attributs
    return (super.equals_(obj) && ...
  }

  public boolean equals(Object x)
  {
    if ((x!=null) && (x.getClass()==Employé.class))
    { return equals_((Employé)x);
    }
    return false;
  }
}
```

Les classes `EmployéCDD` et `Sportif` font de même.

Cette version fonctionne s'il n'existe pas de version souple dans l'arbre d'héritage. Cela ne peut pas être connu a priori. Il faut alors enrichir le code. Cela sera étudié plus loin. Regardons pour le moment comment rédiger la version souple.

2. APPROCHE SOUPLE



La méthode `equals()`, version souple de la classe `Personne` peut être rédigé ainsi :

```
public class Personne
{ // Version souple
  public boolean equals(Object x)
  { if (x instanceof Personne)
    { // Comparaison des attributs
      Personne obj=(Personne)x;
      return ...
    }
    return false;
  }
}
```

La méthode `equals()` de la classe `Employé` doit tenir compte de l'héritage.

```
public class Employé extends Personne
{ // Version souple
  public boolean equals(Object x)
  {
    boolean rc=super.equals(x);
    if ((rc) && (x instanceof Employé))
    { // Comparer les attributs
      Employé obj=(Employé)x;
      return (...
    }
    return rc;
  }
}
```

Les classes `EmployéCDD` et `Sportif` font de même.

Cette version fonctionne s'il n'existe pas de version rigide dans les sous-classes de l'arbre d'héritage. Cette information ne peut pas être connue à priori. Il faut alors enrichir le code.

3. OFFRIR LES DEUX APPROCHES

Pour chaque niveau d'héritage, il est possible de choisir l'approche souple ou rigide. Cela est possible moyennant un certain nombre de contraintes.

- Au même niveau de profondeur de l'héritage, toutes les versions doivent être du même type.
- Sous une classe proposant l'approche rigide, il ne peut y avoir que des approches rigides.

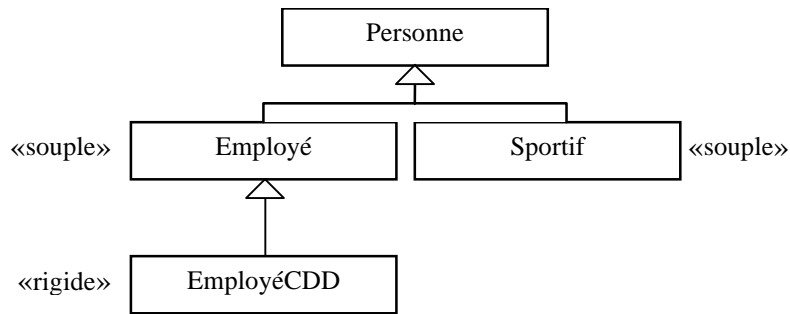
Regardons le premier point : La méthode `equals()` doit être réflexive. C'est-à-dire que l'appel de `a.equals(b)` doit toujours être égal à `b.equals(a)`. Si la classe `Employé` propose une approche souple et la classe `Sportif` propose une approche rigide, il n'y a plus de réflexivité. `employer.equals(sportif)` donne `false`, alors que `sportif.equals(employer)` peut donner `true`. Pour éviter cela, il faut qu'à un niveau de profondeur de l'arbre d'héritage, toutes les versions de la méthode `equals()` utilisent la même stratégie.

D'autre part, si une sous-classe propose une version souple alors que sa super-classe propose une approche rigide, la méthode `equals()` retournera toujours la valeur `false`. Par exemple, si la classe `Employé` propose une version rigide et que la classe `EmployéCDD` propose une version souple, la comparaison d'un `Employé` avec un `EmployéCDD` sera toujours `false`.

Pour respecter ces contraintes, le modèle objet peut par exemple proposer les approches suivantes :

- une approche souple pour un `Employé` ;

- une approche souple pour un `Sportif` ;
- une approche rigide pour un `EmployéCDD`.



Avec les versions actuelles des méthodes `equals()`, il n'est pas possible de tenir compte de cette diversité. L'approche actuelle impose une version souple ou rigide pour tout l'arbre d'héritage. Il faut ajouter le nécessaire pour rendre la méthode `equals()` réflexive quelles que soient les options choisies dans les sous-classes.

Les versions actuelles ne garantissent pas la réflexivité. Pour cela, il faut détecter qui, de l'instance courante ou du paramètre, représente une plus grande spécialisation. Il faut compter le nombre de super classes de chacun et identifier celui qui en possède le plus. `EmployéCDD` connaît `Employé`, mais `Employé` ne doit pas connaître `EmployéCDD`. On peut détecter cela en constatant que `EmployéCDD` possède deux supers classes alors qu'`Employé` n'en possède qu'une.

Ajoutons une méthode statique `CountSuper()` à la classe `Equals`. Cette méthode permet de compter le nombre de super classe.

```

private class Equals
{
  static public int CountSuper(Object obj)
  {
    int i=0;
    for (Class cl=obj.getClass();cl!=null;cl=cl.getSuperclass(),++i);
    return i;
  }
};

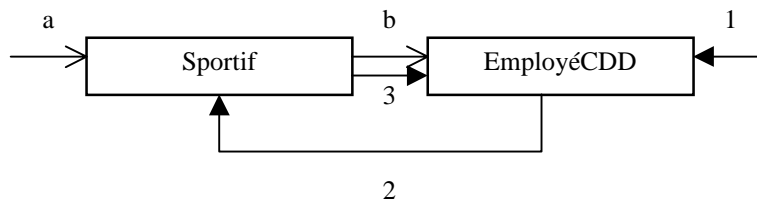
```

Si le paramètre est plus spécialisé que l'instance courante, l'appel de la méthode est inversé.

```

if (Equals.CountSuper(x)>Equals.CountSuper(this))
  return x.equals(this); // Inverse

```



Pour comparer un `Sportif` et un `EmployéCDD`, si le membre à gauche de la comparaison est un sportif, le programme entre par la flèche "a", puis compare ces attributs via la flèche "b". Si le membre à gauche est un `EmployéCDD`, le programme entre par la flèche "1", la méthode détecte que le `Sportif` est plus spécialisé, alors la méthode passe par la flèche "2" pour demander au `Sportif` de s'occuper de la comparaison. Le `Sportif` utilise la flèche "3" pour comparer les attributs.

Ainsi, on utilise toujours la méthode `equals()` la plus spécialisée.

```

public class Personne
{
  public boolean equals(Object x)
  {
    if (x==null) return false;
    if (Equals.CountSuper(x)>Equals.CountSuper(this))
      return x.equals(this); // Inverse
    //...
  }
}

```

Il faut ajouter ce test pour les versions souples et rigides.

4. APPROCHE RIGIDE ETENDUE

La nouvelle version de l'approche rigide intègre l'inversion de l'appel si nécessaire. Le modèle devient :

```

public class Personne
{
  //...
  // Version rigide
  protected final boolean equals_(Personne obj)
  {
    // Comparer les attributs
  }
}

```

```

    return ...
}

public boolean equals(Object x)
{ if (x==null) return false;
  if (Equals.CountSuper(x)>Equals.CountSuper(this))
    return x.equals(this); // Inverse
  if (x.getClass()==Personne.class)
    return equals_((Personne)x);
  return false;
}
}

```

La sous-classe `Employé` devient :

```

public class Employé extends Personne
{ //...
  // Version rigide
  protected final boolean equals_(Employé obj)
  { // Comparer les attributs
    return (super.equals_(obj) && ...
  }

  public boolean equals(Object x)
  {
    if (x==null) return false;
    if (Equals.CountSuper(x)>Equals.CountSuper(this))
      return x.equals(this);
    if (x.getClass()==Employé.class)
      return equals_(Employé)x);
    return false;
  }
}

```

5. APPROCHE SOUPLE ETENDUE

La nouvelle version pour la classe `Personne` intègre l'inversion de l'appel si nécessaire. Le modèle devient :

```

public class Personne
{ //...
  // Version souple
  public boolean equals(Object x)
  { if (x==null) return false;
    if (Equals.CountSuper(x)>Equals.CountSuper(this))
      return x.equals(this); // Inverse
    // Comparaison des attributs
    Personne obj=(Personne)x;
    return ...
  }
}

```

La classe `Employé` doit également détecter l'instance la plus spécialisée.

```

public class Employé extends Personne
{ //...
  // Version souple
  public boolean equals(Object x)
  {
    if (x==null) return false;
    if (Equals.CountSuper(x)>Equals.CountSuper(this))
      return x.equals(this); // Inverse
    boolean rc=super.equals(x);
    if ((rc) && (x instanceof Employé))
    { // Comparer les attributs
      Employé obj=(Employé)x;
      return (...
    }
    return rc;
  }
}

```

6. LE LIEN AVEC LA METHODE

La méthode `hashCode()` permet d'obtenir un résumé d'une instance. Cela permet d'optimiser les algorithmes de recherche dans un ensemble de valeur. Ces algorithmes finissent toujours par appeler la méthode `equals()` pour trouver l'instance recherchée. Il est nécessaire de coder la méthode `hashCode()` si vous proposez la méthode `equals()`. Si vous savez que cette méthode ne doit jamais être utilisée, générez une exception lors de l'appel à `hashCode()`.

```
public class MaClass
{
    public int hashCode()
    { throw new NoSuchMethodException();
    }
    ...
}
```

7. POURQUOI EST-CE SI COMPLEXE ?

Pourquoi cette méthode est-elle si difficile à rédiger ? Java offre le polymorphisme en sélectionnant une méthode par rapport à une instance, celle référencé par la variable à la gauche du point. Il est parfois nécessaire d'avoir un polymorphisme dépendant de plusieurs instances. C'est le cas de la méthode `equals()`. `a.equals(b)` doit être similaire à `b.equals(a)`. Le traitement `equals()` n'est pas une vraie méthode, mais un traitement présent entre deux instances. Certains langages informatiques offrent un polymorphisme multiple. C'est le cas de CLOS. Il est ainsi possible de déclarer des fonctions avec différents types de paramètres et de demander au programme de sélectionner la bonne version lors de l'exécution. Cela pourrait se traduire en java comme ceci :

```
boolean equals(Personne p1, Personne p2) { ... }
boolean equals(Employé p1, Personne p2) { ... }
boolean equals(Personne p1, Employé p2) { ... }
boolean equals(Employé p1, Employé p2) { ... }
...
```

Lors de l'appel à `equals(o1,o2)`, la bonne version est sélectionnée à l'exécution.

Java n'offrant pas de polymorphisme général, il faut faire preuve de rigueur et d'imagination pour contourner le problème. On y arrive parfois. C'est le cas de la méthode `equals()` ci-dessus.

En fait, la méthode `equals()` n'a pas vocation à être polymorphique avec une seule instance. Elle ne devrait pas être présente dans la classe `Object`. Une meilleure approche consiste à déclarer différente version `final` de la méthode `equals()` avec des paramètres différents.

```
public class Personne
{
    public final boolean equals(Personne obj)
    { ...
    }
}

public class Employé extend Personne
{
    public final boolean equals(Employé obj)
    { ...
    }
}
```

Cela permet au compilateur d'optimiser le code généré, et permet d'éviter une partie des utilisations erronées. En effet, l'appel de `equals()` avec un paramètre de type `Date` est ainsi interdit par le compilateur. La version actuelle de `equals()` l'accepte ! Par contre, tout n'est pas résolu. Il faut continuer à vérifier si l'instance du paramètre est du même type que l'instance courante. Ce code n'interdit pas d'appeler la méthode `equals(Personne)` avec un paramètre de type `Employé`.

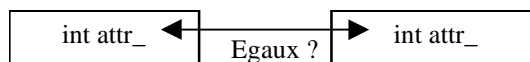
8. COMMENT COMPARER LES ATTRIBUTS ?

Une fois que l'on a décidé quelle stratégie utilisée, il faut comparer tous les attributs. Regardons comment s'occuper de cela.

8.1 Les attributs primitifs

Il n'y a pas de difficulté pour comparer les attributs primitifs. Il suffit d'utiliser l'opérateur `==`.

```
public class MaClass
{ int _attr;
  public boolean equals(Object x)
  { //...
    MaClass obj=(MaClass)x;
    return (_attr==obj._attr)
  }
}
```



8.2 Les relations

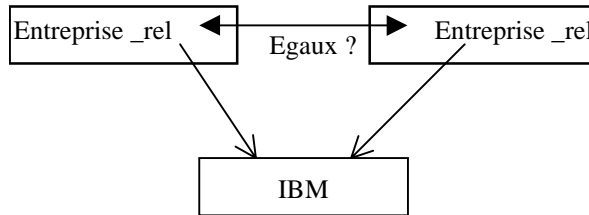
Pour comparer les relations, il faut procéder comme pour les types primitifs, en utilisant l'opérateur `==`.

```
public class MaClass
{ Entreprise _rel;
  public boolean equals(Object x)
```

```

{ //...
  MaClass obj=(MaClass)x;
  return (_rel==obj._rel);
}
}

```



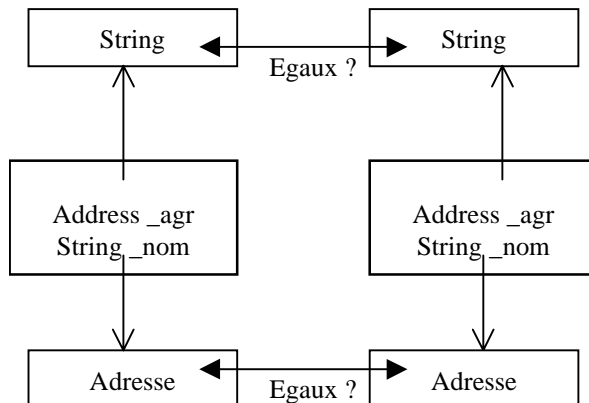
8.3 Les agrégations

Pour comparer des instances agrégées, il faut utiliser la méthode `equals()` de ces instances. La classe `String` est une agrégation. Il faut impérativement utiliser sa méthode `equals()`.

```

public class MaClass
{ Address _agr;
  String _nom;
  public boolean equals(Object x)
  { //...
    MaClass obj=(MaClass)x;
    return (((_agr==obj._agr) && (_agr==null)) ||
            ((_agr!=null) && _agr.equals(obj._agr))
            ) &&
            (((_nom==obj._nom) && (_nom==null)) ||
            ((_nom!=null) && _nom.equals(obj._nom))
            );
  }
}

```



Si l'agrégation est obligatoire, il n'est pas nécessaire de vérifier si les pointeurs sont à `null`.

```

public class MaClass
{ Address _agr;
  String _nom;
  public boolean equals(Object x)
  { //...
    MaClass obj=(MaClass)x;
    return (_agr.equals(obj._agr) &&
            _nom.equals(obj._nom)
            );
  }
}

```

Il est possible d'éviter d'utiliser la méthode `equals()` sur une instance de type `String` à condition que la chaîne soit présente dans le groupe des constantes.

La méthode `String.intern()` permet de placer une chaîne dans le groupe des constantes, ou de retourner une instance déjà présente. Les chaînes constantes sont présentes dans le groupe des constantes (à condition d'utiliser au moins une version 1.1 de Java). Dans ce cas, il est possible de comparer deux chaînes à l'aide de l'opérateur égale.

```

("a"+"b"+"c").intern()=="abc"

```

Si une chaîne doit être construite, mais à une durée de vie longue, il peut être judicieux de la placer dans le groupe de constante. Par exemple,

```

public class MaClass
{ private String name_;

    public MaClass(String name)
    { name_=name.intern();
    }

    boolean equals(Object x)
    { MaClass obj=(MaClass)x;
      return obj.name_.equals(name_);
    }
}

```

Une instance `MaClass` est initialisée à partir d'une chaîne de caractère qui peut éventuellement être construite. Pour garder cette chaîne dans le groupe de constante, la méthode `intern()` est appelée. Ainsi, la méthode `equal()` peut comparer les chaînes sans utiliser la méthode `String.equals()`.

8.4 Les caches

Il ne faut pas vérifier les informations présentes dans des caches. Les caches sont des informations redondantes. Elles sont déductibles des informations principales. Deux instances différentes peuvent avoir les mêmes valeurs principales, mais des valeurs de caches différentes.

8.5 Multitâche

Si une des instances peut être modifiée par une autre tâche, il faut protéger les instances pour un accès concurrent.

```

public class Employé extends Personne
{
    ...
    protected synchronized final boolean equals_(Employé obj)
    { synchronized(obj)
      {
          // Comparaison des attributs
          return (super.equals_(obj) && ...
      }
    }
}

```

Attention, si deux appels simultanés ont lieu sur les mêmes instances, il peut y avoir un risque d'étreinte mutuelle.

```

a.equals_(b)
b.equals_(a)

```

Pour la méthode `like()`, le modèle est celui-ci.

```

public class Employé extends Personne
{
    public boolean like(Personne x)
    { if (x==this) return true;
      boolean rc=super.like(x);
      if ((rc) && (x instanceof Employé))
      { synchronized(this)
        { synchronized(x)
          {
              // Comparaison des attributs
              Employé obj=(Employé)x;
              return (...
          }
        }
      }
      return rc;
    }
}

```

8.6 Optimisation

Il y a plusieurs approches pour optimiser la méthode `equals()`.

8.6.1 Comparer avec `this`

Si le paramètre est égal à `this` la méthode peut retourner immédiatement `true` sans avoir à tester chaque attribut.

```

public boolean equals(Object x)
{ if (this==x) return true;
  ...
}

```

8.6.2 Trier les attributs

Lors de la comparaison des attributs, il est judicieux de commencer par l'attribut ayant le plus de chance d'être différent. Par exemple, pour la classe `Date`, il est préférable de vérifier d'abord le jour, puis le mois et enfin l'année. Dans une application, les dates ont généralement la même année, plus ou moins un an. Par contre, le jour est très variable. Dans ce cas, il faut comparer les attributs comme ceci :

```
public class Date
{
    ...
    public boolean equals(Object x)
    {
        ...
        Date obj=(Date)x;
        return ((_day==obj._day) &&
                (_month==obj._month) &&
                (_year==obj._year));
    }
}
```

Il suffit d'un seul attribut différent pour que la méthode `equals()` retourne `false`. Par contre, il faut que tous les attributs soient égaux pour retourner `true`. Il faut chercher à mettre rapidement en échec la comparaison.

8.6.3 Les instances immuables

La classe `Object` propose la méthode `hashCode()`. Celle-ci doit être redéfinie pour permettre le calcul d'un résumé des valeurs de l'instance. Il peut exister deux instances ayant des attributs différant mais possédant la même valeur de hash, mais il ne peut pas exister deux valeurs de hash pour les mêmes valeurs des attributs.

Pour optimiser la comparaison des instances immuables, il faut rédiger correctement la méthode `hashCode()` et garder le résultat dans un cache. Avant de comparer les attributs un à un, il faut vérifier si la valeur de hash présente dans le cache est identique. Si ce n'est pas le cas, il faut retourner `false`, sinon, il faut vérifier les attributs.

```
public class Immuable
{
    private int _cacheHash;
    public Immuable(...)
    {
        ... // Initialise les attributs
        _cacheHash=... // Calcule le hash
    }
    public int hashCode()
    {
        return _cacheHash;
    }
    public boolean equals(Object x)
    {
        ...
        Immuable obj=(Immuable)x;
        if ((_cacheHash==obj._cacheHash) && // Le hash est bon ?
            ... // Compare les attributs
            return true;
        return false;
    }
}
```

Il ne faut pas comparer la valeur de hash si elle n'est pas dans un cache. En effet, pour calculer le hash, il faut consulter les attributs. Ensuite, il faut de toute façon les consulter pour vérifier l'égalité. La première étape est inutile. Si par contre, la valeur de hash a été précédemment calculer, il est intéressant d'en tenir compte.

8.6.4 Mémoriser la valeur de hash

Il est possible de garder la valeur de hash avec un drapeau indiquant sa pertinence. Les méthodes modifiant l'instance vont baisser le drapeau. La méthode `hashCode()` va calculer la valeur de hash, et lever le drapeau. Ainsi, la méthode `equals()` peut vérifier l'état du drapeau, et utiliser ou non la valeur de hash mémorisé dans l'instance.

```
public class MaClass
{
    private boolean _hashOk=false;
    private int _cacheHash;
    ...
    public void modifieInstance(...)
    {
        _hashOk=false;
        ...
    }
    public int hashCode()
    {
        if (!_hashOk)
        {
            _cacheHash=... // Calcule le hash
            _hashOk=true;
        }
        return _cachHash;
    }
    public boolean equals(Object x)
    {
        ...
        MaClass obj=(MaClass)x;
        if ((_hashOk) && (obj._hashOk))
            return ((_cacheHash==obj._cacheHash)

```

```

        && ... // Compare les attributs
    else return (... // Compare les attributs
    }
}

```

Il faut être rigoureux dans le suivi de la valeur de hash. Toutes les méthodes pouvant entraîner une modification de la valeur de hash doivent mettre le drapeau `_hashOk` à `false`.

9. CONCLUSION

Pour simplifier cela, il faut différencier clairement les deux approches. La méthode `equals()` doit proposer toujours l'approche rigide en utilisant la première approche (sans inversion possible de l'appel). Pour proposer l'approche souple, il faut utiliser une méthode appelée `like()` implantant la première approche souple.

```

public class Personne
{
    protected final boolean equals_(Personne obj)
    { // Comparaison des attributs
        return (...
    }

    public boolean equals(Object x)
    { if (x==this) return true;
      if ((x!=null) && (x.getClass()==Personne.class))
        { return equals_((Personne)x);
        }
      return false;
    }

    public boolean like(Personne x)
    { if (x==this) return true;
      if (x instanceof Personne)
        { // Comparaison des attributs
          Personne obj=(Personne)x;
          return ...
        }
      return false;
    }
}

```

Pour les sous-classes, nous avons :

```

public class Employé extends Personne
{
    private int salaire_;

    public Employé(String nom,String prenom,int salaire)
    { super(nom,prenom);
      salaire_=salaire;
    }

    protected synchronized final boolean equals_(Employé obj)
    {
        synchronized(obj)
        { // Comparaison des attributs
            return (super.equals_(obj) && ...
        }
    }

    public boolean equals(Object x)
    { if (x==this) return true;
      if ((x!=null) && (x.getClass()==Employé.class))
        { return equals_((Employé)x);
        }
      return false;
    }

    public boolean like(Personne x)
    { if (x==this) return true;
      boolean rc=super.like(x);
      if ((rc) && (x instanceof Employé))
        { synchronized(this)
          { synchronized(x)
            {
                // Comparaison des attributs
                Employé obj=(Employé)x;
                return (...
            }
          }
        }
    }
}

```

```
    return rc;  
}
```

La méthode `equals()` est très utile pour comparer deux instances. Cela permet par exemple de vérifier à la fin d'une boîte de dialogue si l'utilisateur a apporté des modifications sur les objets métiers. Seulement dans ce cas, la base de données est mise à jour. La classe `Dictionary` utilise la méthode `equals()` pour manipuler les clefs du dictionnaire. Il est important de coder ces méthodes en respectant les modèles ci-dessus.