

Énumération en java

Philippe PRADOS

pp@philippe.prados.name



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

TABLE DES MATIERES

1.1	Cas du <code>null</code>	4
1.2	Conversions	4
1.3	Classe interne	5

Avant-propos

Comment proposer une énumération en Java ? L'énumération est un nouveau type primitif qui possède un ensemble de valeurs discrètes, dont l'usage peut être vérifié par le compilateur.

Le C ou le C++ proposent une syntaxe permettant de décrire une énumération. Cela permet de garantir une utilisation correcte d'un type.

```
enum when { Before, Now, After};
```

Il n'est pas possible de valoriser une énumération avec une valeur ne faisant pas partie des valeurs autorisées. Cela est rejeté par le compilateur.

Java ne propose pas cela. Habituellement, le développeur déclare un ensemble de constantes numériques, et utilise un entier pour représenter l'énumération.

```
public class MyClass
{
    static public final int Before=0;
    static public final int Now=1;
    static public final int After=2;
    private int when_=Before;
    //...
}
```

Parfois, les constantes sont indiquées dans une interface. Les attributs primitifs de type `final` sont utilisés en ligne lors de la compilation.

```
class MyClass
{
    static final int SIZE=3;
    public void f()
    {
        int s=SIZE;
        ...
    }
}
```

La méthode `f()` est traduite par le compilateur comme ceci :

```
public void f()
{
    int s=3;
    ...
}
```

C'est le cas même si l'attribut `final` est présent dans une autre classe ou dans une interface. Cela peut avoir une conséquence fâcheuse. Si vous modifiez la valeur de la constante `SIZE`, sans recompiler toutes les classes qui l'utilise, le programme sera erroné. Ce problème se rencontre essentiellement lors de l'intégration de nouvelle version de librairie. Par exemple, lors du passage d'une version 1.0 à 2.0.

Par contre, si l'attribut `final` est un objet, il y a consultation de l'attribut. Il n'y a alors plus de problème d'évolution de la librairie.

Reprenons l'exemple initial. Si l'utilisateur valorise l'attribut `when_` avec la valeur `3` par exemple, le compilateur ne dit rien. Une méthode recevant un moment en paramètre devra vérifier celui-ci.

```
public class MyClass
{ //...
    public void setWhen(int x)
    {
        if ((x<Before) || (x>After))
            throw new IllegalArgumentException("Invalid parameter");
        when_=x;
    }
}
```

Comment éviter ces tests ? En interdisant syntaxiquement la rédaction d'un code erroné.

Généralement, les énumérations n'ont pas de correspondance numérique. Les nombres choisis pour représenter les valeurs de l'énumération sont arbitraires. Nous désirons identifier les trois moments, mais pas avoir une correspondance numérique entre le *after* et la valeur 2. Il est préférable d'utiliser une classe représentant l'énumération. Cette classe aura un constructeur privé. Les différentes valeurs de l'énumération seront représentées par des instances statiques particulières.

```
public final class when
{
    private when() {}
    public static final when Before=new when();
    public static final when Now    =new when();
    public static final when After  =new when();
}

public class MyClass
{
    private when when_=when.Before;
```

```

public void setWhen(when x)
{
    when_=x;
}
}

```

`Before` est une valeur unique d'un pointeur. `Now` et `After` sont d'autres valeurs pour un pointeur. Le type `when` doit être utilisé avec une sémantique par valeur. On l'utilise comme s'il s'agissait d'un entier. Pour renforcer la similitude avec un type primitif, le nom de la classe commence par une minuscule. Il ne faut pas voir les références de ce type comme des pointeurs, mais comme des valeurs, au même titre qu'un entier. Pour comparer deux variables de type `when` il faut utiliser la comparaison d'identité.

```
if (when1==when2) ...
```

Un pointeur `when` ne peut pointer que sur les trois instances `Before`, `Now` et `After`. En effet, il n'est pas possible de construire d'autres instances `when` car le constructeur est privé. Les vérifications des paramètres ne sont plus nécessaires car l'utilisateur de ce nouveau type ne peut pas offrir une valeur différente des trois disponibles. Pour avoir une variable du type de l'énumération, il faut utiliser le type `when`.

```

public static void main(String[] args)
{
    when w=when.Before;
    //...
    if (w==when.After) ...
    //...
    w=4; // erreur, ne compile pas.
}

```

1.1 Cas du null

La seule situation qui n'est pas gérée par cette approche est la possibilité à une référence `when` d'avoir la valeur `null`. Pour lever ce point, `null` peut faire partie des énumérations.

```

public class when
{ //...
    public static final when Before=null;
    public static final when Now    =new when();
    public static final when After  =new when();
    //...
}

```

`Before` devient la valeur par défaut de l'énumération, au même titre que zéro et la valeur par défaut d'un entier.

```

public class MyClass
{
    int    i_;    // =0 par défaut
    when  when_; // =Before par défaut.
};

```

Certains conteneurs, comme les dictionnaires de java, n'acceptent pas la valeur `null` comme clef ou comme valeur. Si une énumération doit être placée dans un de ces conteneurs, il peut être préférable de ne pas proposer `null` comme valeur possible de l'énumération. Une autre approche consiste à utiliser une classe de service encapsulant l'énumération. C'est ce qu'il faut faire lorsque vous désirez ajouter un entier à un dictionnaire. Vous devez encapsuler le type primitif par la classe `Integer`. L'énumération devant être vue comme un type primitif, il faut ajouter la classe `When` pour transformer une valeur `when` en objet.

```

public final class When
{
    when when_;
    public When(when w)
    {
        when_=w;
    }
    public when whenValue()
    {
        return when_;
    }
}

```

Une instance de type `When` peut servir de clef à un dictionnaire.

1.2 Conversions

La classe `When` permet de proposer un objet pour le type *pseudo* primitif `when`. Comme pour la classe `Integer`, la classe `When` peut proposer une méthode statique `toString()` pour retourner une version textuelle des différentes valeurs possibles.

```

public class When
{ //...
    public static String toString(when x)
    {
        if      (x==when.Before) return "Before";
        else if (x==when.Now)    return "Now";
        else                      return "After";
    }
}

```

```

}
}

```

Si l'énumération doit pouvoir être convertie en valeur numérique, deux approches sont alors possibles. Si `null` est une valeur possible de l'énumération, il suffit d'ajouter une méthode statique `toInt()` à la classe `When`.

```

public class When
{ //...
    public static int toInt(when x)
    {
        if (x==when.Before) return 0;
        else if (x==when.Now) return 1;
        else return 2;
    }
    //...
}

```

Si `null` n'est pas une valeur possible de l'énumération, des méthodes d'instance `toInt()` et `toString()` peuvent être proposées.

```

public final class when
{
    private int val_;
    private when(int val)
    {
        val_=val;
    }
    public int toInt()
    {
        return val_;
    }
    public String toString()
    {
        if (this==when.Before) return "Before";
        else if (this==when.Now) return "Now";
        else return "After";
    }
    public static final when Before=new when(0);
    public static final when Now =new when(1);
    public static final when After =new when(2);
}

```

1.3 Classe interne

Si l'énumération ne peut être utilisée que dans le contexte d'une classe, il peut être intéressant de proposer le type `when` en tant que classe interne. De même pour la classe `When`.

```

public class MyClass
{
    public static final class when
    {
        private when() {}
        public static final when Before=null;
        public static final when Now =new when();
        public static final when After =new when();
    }
    public static final class When
    {
        when when_;
        public When(when w)
        {
            when_=w;
        }
        public when whenValue()
        {
            return when_;
        }
        public static String toString(when x)
        { //...
        }
        public static int toInt(when x)
        { //...
        }
    }
}
//...
}
//..
MyClass.when when_=MyClass.when.Now;

```

Les différentes valeurs de l'énumération peuvent être déclarées à l'extérieur de la classe `when`. Cela permet de réduire la taille du nom d'une valeur et de proposer les méthodes statiques `toString()` et `toInt()`..

```
public class MyClass
{
    public static final class when
    {
        private when() {}
    }
    public static final when Before=null;
    public static final when Now    =new when();
    public static final when After =new when();

    public static String toString(when x)
    { //...
    }
    public static int toInt(when x)
    { //...
    }
}
//..
MyClass.when when_=MyClass.Now;
```

Ainsi, `MyClass.when.Now` devient `MyClass.Now`. De même, `MyClass.When.toString()` devient `MyClass.toString()`.

Il est préférable de déclarer une énumération dans une classe externe et d'accepter la valeur `null`. Cela déclare un nouveau type *pseudo* primitif qui peut être utilisé partout, sans risque d'erreur. Le compilateur fait automatiquement toutes les vérifications nécessaires. De plus, cela garantit une évolution sans heurt des bibliothèques.