# Introduction aux énumérations en C#

par François Guillot (fguillot.developpez.com)

Date de publication : 2011-01-12

Dernière mise à jour :

Quoi de plus simple en apparence que les énumérations ? En apparence, car elles sont bien plus qu'une simple liste de constantes et leur fonctionnement, ainsi que certains mécanismes qui leur sont propres méritent que l'on s'y attarde.

Cet article se propose de présenter leur fonctionnement, d'exposer leurs subtilités et de montrer un aperçu de leurs possibles utilisations.

# I - Les énumérations

## I-A - Présentation

Une énumération permet de regrouper une liste de constantes nommées, de cette façon :

```
public enum FrenchDayOfWeek
{
    Lundi,
    Mardi,
    Mercredi,
    Jeudi,
    Vendredi,
    Samedi,
    Dimanche
}
```

Chacune de ces valeurs représente une constante. Elles obéissent aux mêmes règles de nommage que les variables : pas de chiffre au début, pas d'espace, pas de tiret ainsi qu'un certain nombre d'autres signes.

L'avantage immédiat d'une énumération est de pouvoir s'affranchir de l'utilisation de variables volatiles afin de faire circuler des indications dans votre programme.

Considérez le code suivant :

```
class Program
{
    static void Main(string[] args)
    {
        const string todaysDay = "samdi";
        if (IsWeekendDay(todaysDay))
            Console.WriteLine("Youpi c'est le week-end !");
        else
            Console.WriteLine("Métro boulot dodo...");
    }

    private static bool IsWeekendDay(string day)
    {
        if (day.ToLower() == "samedi" || day.ToLower() == "dimanche")
            return true;

        return false;
    }
}
```

Le développeur a créé une constante indiquant un jour de la semaine sous forme de chaîne de caractères. Mais pas de bol, il a fait une faute de frappe et a écrit « samdi ». Il teste ensuite si ce jour correspond à un jour de week-end, et comme vous pouvez le voir, la réponse sera... non, car la fonction IsWeekendDay est incapable de juger qu'il y a une faute.

Bien sûr, on aurait pu mettre des contrôles pour s'assurer que le jour était dans un format cohérent, mais pourquoi se compliquer la vie ? Les énumérations sont là pour ça :

```
class Program
{
    static void Main(string[] args)
    {
        const FrenchDayOfWeek todaysDay = FrenchDayOfWeek.Samedi;
        if (IsWeekendDay(todaysDay))
            Console.WriteLine("Youpi c'est le week-end !");
        else
            Console.WriteLine("Métro boulot dodo...");
    }
```

```
    private static bool IsWeekendDay(FrenchDayOfWeek day)
    {
        if (day == FrenchDayOfWeek.Samedi || day == FrenchDayOfWeek.Dimanche)
            return true;

        return false;
    }
}
```

Grâce à l'énumération FrenchDayOfWeek, plus de risque de faute de frappe, ou plus généralement de ce qu'on appelle communément les « magic string ». En effet les différentes valeurs d'une énumération sont fixes, et une faute de frappe lors de la saisie d'une de ces valeurs entraînerait l'impossibilité de pouvoir compiler le programme.

Les énumérations ne servent pas seulement à faire circuler une indication, elles sont aussi très utiles lorsque l'on veut en récupérer une. Par exemple, imaginons que vous souhaitiez savoir si un utilisateur a le droit de se connecter à votre application :

```
class Program
{
    static void Main(string[] args)
    {
        if (UserCanConnect("foo", "bar"))
            Console.WriteLine("Connexion réussie !");
        else
            Console.WriteLine("Connexion refusée.");
    }

    private static bool UserCanConnect(string login, string password)
    {
        if (UserExist(login) && LoginAndPasswordMatch(login, password))
            return true;

        return false;
    }
}
```

La fonction UserCanConnect détermine si les paramètres de connexion (login et mot de passe) d'un utilisateur sont justes ; elle renvoie un booléen pour indiquer le résultat. Le problème c'est qu'avec ce résultat il est impossible de savoir - le cas échéant - pourquoi l'utilisateur n'est pas en mesure de se connecter. En utilisant une énumération, on apporte un niveau de précision supplémentaire :

```
public enum ConnectionResult
{
    ConnectionOK,
    WrongLogin,
    WrongPassword
}

class Program
{
    static void Main(string[] args)
    {
        ConnectionResult connectionResult = UserCanConnect("foo", "bar");
        if (connectionResult == ConnectionResult.ConnectionOK)
            Console.WriteLine("Connexion réussie !");
        else if (connectionResult == ConnectionResult.WrongLogin)
            Console.WriteLine("Connexion refusée : login inconnu.");
        else if (connectionResult == ConnectionResult.WrongPassword)
            Console.WriteLine("Connexion refusée : mot de passe erroné.");
    }

    private static ConnectionResult UserCanConnect(string login, string password)
    {
        if (UserExist(login))
        {
            if (LoginAndPasswordMatch(login, password))
```

```
                return ConnectionResult.ConnectionOK;
            else
                return ConnectionResult.WrongPassword;
        }

        return ConnectionResult.WrongLogin;
    }
}
```

Maintenant nous pouvons savoir non seulement si l'utilisateur a le droit de se connecter, mais aussi pourquoi la connexion peut lui être refusée.

Remarquez que pour déclarer une énumération, on utilise le mot-clef « enum ». Les énumérations étant des « first-class citizens », leur déclaration crée un type à part entière, et peut se faire aussi bien au niveau d'un espace de nom que comme membre d'une classe. En fonction de l'un ou l'autre, les modificateurs d'accès que l'on pourra leur appliquer varieront aussi, mais seront les mêmes que pour une classe.

Ces différentes possibilités rendent les énumérations à la fois très utiles et extrêmement simples à manipuler. Pourtant il y a quelques faits remarquables les concernant qui méritent que l'on s'y attarde.

## I-B - L'énumération, un type valeur

Il existe deux types d'objets dans le Framework .NET : les types valeur et les types référence. Une énumération est un type valeur, comme le sont int, float, bool, char, DateTime et de nombreux autres (mais pas string !). Cela signifie qu'elle est instanciée sur la pile (en anglais : « stack »), et que lorsqu'on la passe en paramètre à une fonction, une copie de sa valeur est effectuée de telle sorte que la fonction n'a pas accès à la copie originale (à moins d'utiliser explicitement le mot-clef « ref »).

Mais plus important encore, comme pour tous les types valeur, la taille d'une énumération en mémoire est fixe. C'est un gage de performance, puisque cela signifie que l'allocation et la désallocation de mémoire allouée à l'énumération se font sans passer par le ramasse-miettes.

Tout cela est possible parce que lors de la compilation, une énumération est générée sous la forme d'une structure, et que ses différentes constantes sont remplacées par ce qu'on appelle les « types intégraux ». La contrepartie est que, puisque la génération se fait une fois pour toutes à la compilation, l'énumération ne pourra en aucun cas être modifiée à l'exécution, même par réflexion. Il est en revanche possible de générer à l'exécution du code contenant la définition d'une énumération, et de compiler celui-ci à la volée.

## I-C - Le type intégral sous-jacent

Comme nous l'avons vu, lors de la compilation les constantes exposées par une énumération sont remplacées par des types intégraux. Cela ne se fait pas au hasard, et il vous est même possible de préciser quel est ce type intégral sous-jacent dans la déclaration de votre énumération.

Justement, reprenons notre énumération FrenchDayOfWeek de tout à l'heure :

```
public enum FrenchDayOfWeek : int
{
    Lundi = 0,
    Mardi = 1,
    Mercredi = 2,
    Jeudi = 3,
    Vendredi = 4,
    Samedi = 5,
    Dimanche = 6
}
```

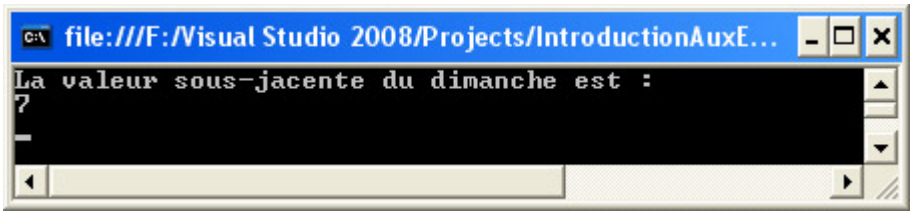
Voilà très précisément ce que voit le compilateur dans cette déclaration. En effet, par défaut le type intégral sous-jacent d'une énumération est le type int, et à chaque constante de l'énumération correspond une valeur qui, toujours par défaut, commencera par zéro et s'incrémentera d'une unité à chaque nouvelle constante.

*Les types intégraux autorisés sont les suivants : int, short, long, uint, ushort, ulong, byte et sbyte.*
*Cela correspond à la liste des types intégraux standards du Framework .NET.*
*Notez bien que les types valeur décimaux comme float, ou d'autres types valeur comme char ou bool en sont exclus.*

Possibilité nous est offerte de personnaliser le type sous-jacent d'une énumération ou ses valeurs. Dans notre exemple, le nombre entier associé à chaque constante ne paraît pas très cohérent, il serait donc judicieux de le modifier, et au passage nous pourrions changer de type sous-jacent pour le type short :

```
public enum FrenchDayOfWeek : short
{
    Lundi = 1,
    Mardi,
    Mercredi,
    Jeudi,
    Vendredi,
    Samedi,
    Dimanche
}

class Program
{
    static void Main(string[] args)
    {
        short valueOfSunday = (short)FrenchDayOfWeek.Dimanche;
        Console.WriteLine("La valeur sous-jacente du dimanche est :");
        Console.WriteLine(valueOfSunday);
    }
}
```




Remarquez qu'il suffit d'indiquer que nous souhaitons que la valeur de la première constante soit "1", pour que toutes celles qui suivent se voient automatiquement incrémentées d'une unité. Ici la valeur de Dimanche est maintenant sept. La contrepartie de ce mécanisme est que vous devez toujours vous méfier de l'ordre dans lequel vous déclarez vos constantes dans l'énumération, en particulier si vous comptez vous servir des valeurs qui leur sont associées.

Remarquez aussi que pour obtenir la valeur d'une des constantes d'une énumération, il suffit de la caster explicitement vers le type sous-jacent de celle-ci, comme nous le faisons ici pour initialiser la valeur de la variable valueOfSunday.

Sachez aussi qu'il est possible de calculer la valeur d'une constante d'une énumération à partir d'une autre constante, que celle-ci soit incluse dans l'énumération ou extérieure à celle-ci, du moment qu'elle est définie à la compilation :

```
public enum SomeEnum
{
    FirstConstant = 99,
    SecondConstant = FirstConstant + 1, // 100
    ThirdConstant = AnyClass.SOME_CONSTANT // 200
}

public class SomeClass
{
```

```
    public const int SOME_CONSTANT = 200;
}
```

Vous pouvez en outre donner à plusieurs constantes de l'énumération la même valeur :

```
public enum ErrorCode
{
    FatalError = 1,
    DeadlyError = 1,
    NonRecoverableError = 1,
    ScreenOfDeath = 2,
    DeadLock = 2,
    AppHasFrozen = 2
}
```

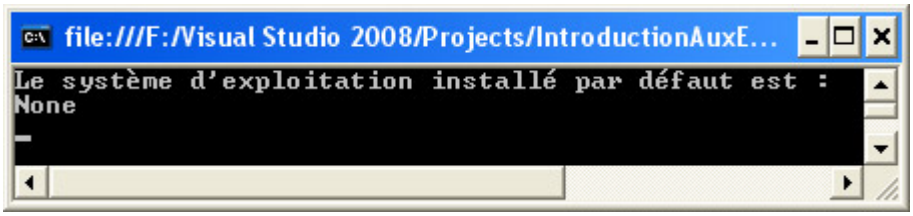
## I-D - La valeur par défaut et les valeurs fantômes

Dans une énumération, la constante nommée dont la valeur est zéro fait office de valeur par défaut. Il est donc préférable - quoique non obligatoire - de toujours conserver une constante dont la valeur soit zéro.

Observez le résultat du code suivant dans lequel - chose inhabituelle - nous utilisons le constructeur par défaut de l'énumération :

```
public enum InstalledOS
{
    Windows = 1,
    MacOS = 2,
    Linux = 3,
    None = 0
}

class Program
{
    static void Main(string[] args)
    {
        InstalledOS os = new InstalledOS();
        Console.WriteLine("Le système d'exploitation installé par défaut est :");
        Console.WriteLine(os);
    }
}
```

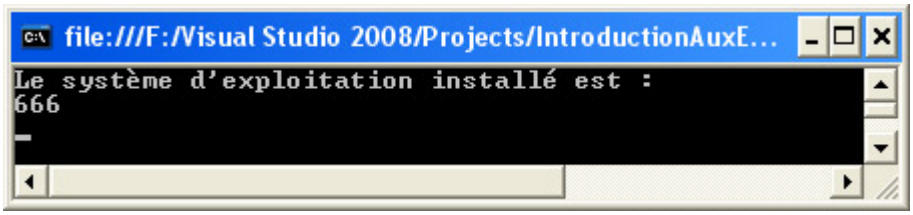



Ce code met en lumière l'existence de notre valeur par défaut. Mais que se serait-il passé si aucune des constantes nommées de notre énumération n'avait eu la valeur zéro ? Je vous fais grâce de la démonstration, mais sachez que ce même code aurait retourné... le chiffre zéro, sans lever la moindre exception.

En fait, une énumération contient virtuellement toutes les valeurs possibles de son type sous-jacent. Pour int, le type sous-jacent par défaut, il s'agira donc de la plage de nombres allant de -2147483648 à +2147483647.

Il est donc possible, avec un peu d'astuce, de déclarer une variable du type d'une énumération, et de lui donner une valeur qui n'est - à priori - pas proposée par ladite énumération. Démonstration :

```
InstalledOS os = (InstalledOS)Enum.Parse(typeof(InstalledOS), "666");
Console.WriteLine("Le système d'exploitation installé est :");
Console.WriteLine(os);
```




Notez que l'énumération InstalledOS n'a pas été modifiée entre cet exemple et l'exemple précédent. Pourtant, nous avons été en mesure, grâce à la méthode statique Enum.Parse que nous étudierons plus tard, de créer une variable dont la valeur ne fait pas partie de celles qui sont définies dans l'énumération. Cette manipulation est possible car le nombre "666" est bien inclus dans l'étendue couverte par le type intégral int, et donc, fait partie des valeurs attribuables à une variable du type de l'énumération InstalledOS.

*Les énumérations sont beaucoup utilisées dans des switch ou des if/else afin de gérer les différents embranchements d'un programme. Comme on l'a vu ici, il est judicieux de toujours partir du principe qu'une variable du type d'une énumération peut contenir une valeur que celle-ci ne définit pourtant pas parmi ses constantes. Dans cette éventualité, prévoyez donc toujours un cas par défaut, en particulier dans les applications où la sécurité et la stabilité sont critiques.*

## II - La classe Enum

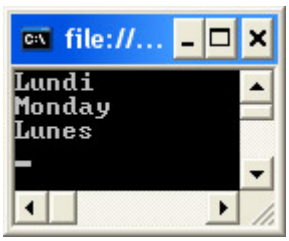
Dans la première partie, nous avons présenté les énumérations et leur mode de fonctionnement. Nous allons maintenant voir les différentes possibilités qui nous sont données pour les manipuler.

Toutes ces possibilités sont regroupées dans la classe Enum. Le « E » majuscule est important car il permet de la différencier du mot-clef enum qui permet de déclarer une énumération proprement dite.

Enum est une classe dont héritent implicitement toutes les énumérations. Son premier intérêt est de permettre grâce au polymorphisme de manipuler des collections de constantes d'énumérations :

```
IList<Enum> enums = new List<Enum>
                        {
                            FrenchDayOfWeek.Lundi,
                            EnglishDayOfWeek.Monday,
                            SpanishDayOfWeek.Lunes
                        };

foreach (Enum item in enums)
    Console.WriteLine(item);
```




Mais comme nous allons le voir maintenant, ce qui est vraiment intéressant avec la classe Enum, ce sont ses méthodes statiques.

### II-A - Obtenir la liste des constantes nommées et des valeurs

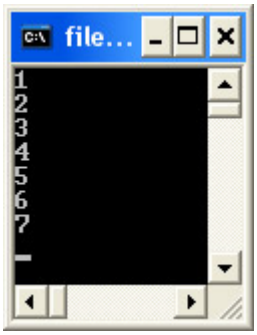
Il est souvent utile de pouvoir récupérer dynamiquement la liste de toutes les constantes incluses dans une énumération, notamment lorsqu'on veut s'en servir pour proposer un choix à l'utilisateur.

Voici comment récupérer la liste des constantes nommées :

```
string[] namedConstants = Enum.GetNames(typeof(FrenchDayOfWeek));
foreach (string constant in namedConstants)
    Console.WriteLine(constant);
```


Voici comment récupérer la liste des valeurs incluses dans l'énumération :

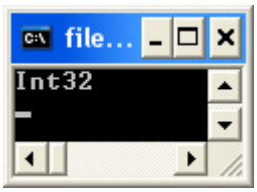
```
Array values = Enum.GetValues(typeof(FrenchDayOfWeek));
foreach (int constant in values)
    Console.WriteLine(constant);
```




On voit que dans le premier cas, la liste des constantes nommées, le tableau retourné est fortement typé. Dans le second, la liste des valeurs, c'est un tableau non typé qui est retourné, et il vous incombe donc de faire un casting vers le type sous-jacent de l'énumération.

Et si on ne connaît pas ce type sous-jacent ? Il est facile de le retrouver grâce à une autre méthode statique de la classe Enum, GetUnderlyingType :

```
Type underlyingType = Enum.GetUnderlyingType(typeof(FrenchDayOfWeek));
Console.WriteLine(underlyingType.Name);
```




Ensuite pour récupérer la liste des valeurs à partir de ce type il suffit de faire :

```
Type underlyingType = Enum.GetUnderlyingType(typeof(FrenchDayOfWeek));
Array values = Enum.GetValues(typeof(FrenchDayOfWeek));
foreach (object constant in values)
    Console.WriteLine(Convert.ChangeType(constant, underlyingType));
```

## II-B - Obtenir le nom ou la valeur d'une constante

Il existe de nombreuses manières d'effectuer cette opération.

Par exemple, pour récupérer le nom d'une constante nommée il suffit d'appeler ToString.

```
string mondayAsString = FrenchDayOfWeek.Lundi.ToString(); // "Lundi"
```

Si l'on souhaite récupérer le nom d'une constante nommée, mais qu'on ne dispose que de sa valeur, on peut utiliser GetName :

```
const int valueOfDay = 4;
string thursdayAsString = Enum.GetName(typeof(FrenchDayOfWeek), valueOfDay); // "Jeudi"

const FrenchDayOfWeek friday = FrenchDayOfWeek.Vendredi;
string fridayAsString = Enum.GetName(typeof(FrenchDayOfWeek), friday); // "Vendredi"
```

GetName accepte en second paramètre soit une variable du type sous-jacent de l'énumération, soit directement une constante nommée de celle-ci ; toute autre valeur entraînera la levée d'une exception. En outre, si la valeur passée en paramètre n'est pas incluse dans l'énumération comme le sont les valeurs fantômes, la chaîne de caractères résultant de l'opération aura la valeur null.

Enfin comme on l'a déjà vu, pour récupérer la valeur d'une constante, il suffit de la caster explicitement vers le type sous-jacent de l'énumération, ou tout autre type compatible :

```
byte tuesdayAsByte = (byte)FrenchDayOfWeek.Mardi; // 2
```

Notez que l'opération inverse est également possible :

```
FrenchDayOfWeek frenchWednesday = (FrenchDayOfWeek)3; // Mercredi
```

## II-C - Convertir une chaîne de caractères en valeur d'énumération

Admettons que vous disposiez de la chaîne de caractères "Mercredi" ou du nombre entier "3", et que vous souhaitiez récupérer une constante d'énumération. Pour cela, il faut utiliser la méthode Parse :

```
const string wedsnedayName = "Mercredi";
FrenchDayOfWeek result1 = (FrenchDayOfWeek)Enum.Parse(typeof(FrenchDayOfWeek), wedsnedayName);

const string wednesdayValue = "3";
FrenchDayOfWeek result2 = (FrenchDayOfWeek)Enum.Parse(typeof(FrenchDayOfWeek), wednesdayValue);
```

On voit ici que la méthode Parse accepte aussi bien une chaîne de caractères représentant le nom d'une constante de l'énumération que l'une de ses valeurs, ce qui s'avère assez pratique. Par contre Parse retourne un simple object, et il vous incombe donc de caster le résultat vers le type de l'énumération. Si vous êtes amené à effectuer souvent cette conversion, et que vous voulez simplifier votre code, vous pouvez utiliser la méthode d'extension générique suivante :

```
public static class EnumTools
{
    public static T ParseToEnum<T>(this string value)
        where T : struct
    {
        return (T)Enum.Parse(typeof(T), value);
    }
}

class Program
{
    static void Main(string[] args)
```

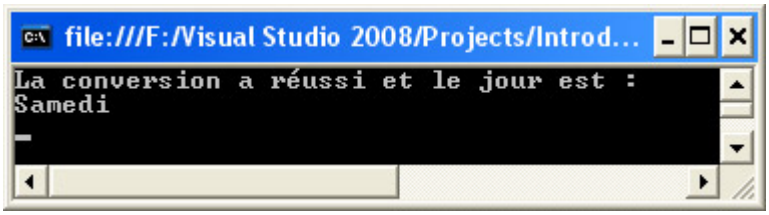
```
    {
        const string thursdayAsInt = "4";
        FrenchDayOfWeek result = thursdayAsInt.ParseToEnum<FrenchDayOfWeek>();
        // result contient maintenant Jeudi
    }
}
```

*Attention ! Si vous passez à la méthode Parse une chaîne de caractères représentant une valeur non incluse dans l'énumération, mais qui entre bien dans la plage nombres couverte par son type sous-jacent, vous créerez une des valeurs fantômes dont nous avons parlé plus haut.*

```
const string devilNumber = "666";
FrenchDayOfWeek result = (FrenchDayOfWeek)Enum.Parse(typeof(FrenchDayOfWeek), devilNumber);
// result contient maintenant une valeur qui ne correspond à aucun jour de la semaine
```

Si vous êtes familier du Framework .NET et du C#, vous saurez que Parse convertit directement la valeur passée en paramètre, et lève une exception en cas d'échec. Le Framework .NET 4.0 a introduit la nouvelle méthode TryParse, déjà présente pour de nombreux types standards comme int ou string, mais qui manquait à la classe Enum. Grâce à elle, vous pouvez tenter de convertir une chaîne de caractères en constante d'énumération, et savoir si l'opération a réussi grâce au booléen qu'elle retourne. Si l'opération a réussi, la constante d'énumération obtenue sera placée dans le paramètre de sortie que vous aurez indiqué avec le mot-clef « out ».

```
FrenchDayOfWeek anotherDay;
if (Enum.TryParse("6", out anotherDay))
{
    Console.WriteLine("La conversion a réussi et le jour est :");
    Console.WriteLine(anotherDay);
}
```




## II-D - Vérifier si l'énumération contient bien un certain membre

L'existence des valeurs fantômes peut vite s'avérer problématique dans une application ayant beaucoup recours aux énumérations. Heureusement une autre méthode statique de la classe Enum permet de vérifier si une certaine constante ou une certaine valeur existe bel et bien au sein d'une énumération, la méthode IsDefined :

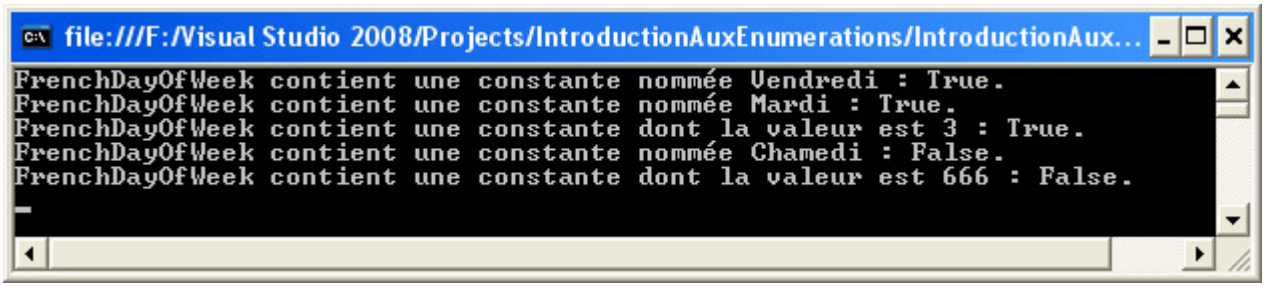
```
bool result1 = Enum.IsDefined(typeof(FrenchDayOfWeek), FrenchDayOfWeek.Vendredi);
Console.WriteLine("FrenchDayOfWeek contient une constante nommée Vendredi : {0}.", result1);

bool result2 = Enum.IsDefined(typeof(FrenchDayOfWeek), "Mardi");
Console.WriteLine("FrenchDayOfWeek contient une constante nommée Mardi : {0}.", result2);

bool result3 = Enum.IsDefined(typeof(FrenchDayOfWeek), 3);
Console.WriteLine("FrenchDayOfWeek contient une constante dont la valeur est 3 : {0}.", result3);

bool result4 = Enum.IsDefined(typeof(FrenchDayOfWeek), "Chamedi");
Console.WriteLine("FrenchDayOfWeek contient une constante nommée Chamedi : {0}.", result4);

bool result5 = Enum.IsDefined(typeof(FrenchDayOfWeek), 666);
Console.WriteLine("FrenchDayOfWeek contient une constante dont la valeur est 666 : {0}.", result5);
```




Comme on le voit IsDefined accepte en second paramètre aussi bien une constante nommée de l'énumération, qu'une valeur de son type sous-jacent, ou encore une chaîne de caractères représentant le nom d'une des constantes (mais pas une valeur de l'une de celles-ci).

## III - L'attribut Flags

Dans une énumération classique, la logique veut que les constantes s'excluent les unes les autres. Pour reprendre notre exemple, on ne peut pas être à la fois lundi et vendredi.

Mais dans certains cas de figure il peut s'avérer utile de pouvoir combiner plusieurs constantes d'une même énumération. C'est à ça que sert l'attribut Flags.

### III-A - Fonctionnement

Prenons un exemple : dans les systèmes Linux, pour attribuer des droits sur les fichiers aux utilisateurs, on utilise un système de champ de bits au format octal. Le tableau suivant résume ce système :

| Permission | Chiffre | Champ de bits |
|---|---|---|
| Exécuter | 1 | 001 |
| Écrire | 2 | 010 |
| Lire | 4 | 100 |

Ainsi, si l'on souhaite qu'un utilisateur ait les droits d'écriture sur un fichier, on lui attribuera le chiffre deux. Mais ces valeurs peuvent aussi être combinées entre elles. Par exemple, pour qu'un utilisateur dispose des droits d'exécution et de lecture sur un fichier, on lui attribuera le chiffre "5", c'est-à-dire "1 + 4", soit le champ de bits "101". De la même façon, si l'on souhaite qu'il dispose de tous les droits (exécution, écriture et lecture), on lui attribuera le chiffre "7", c'est-à-dire "1 + 2 + 4", soit le champ de bits "111".

Ce principe de fonctionnement peut être représenté par l'énumération suivante :

```
[Flags]
public enum LinuxFileAccess
{
    Execute = 1,
    Write = 2,
    Read = 4
}
```

Remarquez que nous avons décoré l'énumération avec l'attribut Flags qui est le sujet de ce chapitre. Cet attribut va permettre d'indiquer au compilateur que l'énumération peut être manipulée comme un champ de bits. Remarquez aussi que pour que ce principe fonctionne, il est nécessaire que les valeurs des constantes soient définies manuellement et que celles-ci s'incrémentent progressivement à la puissance de deux, c'est-à-dire un, puis deux, puis quatre, puis huit, etc.

## III-B - Les opérations sur les champs de bits

Afin de combiner les différentes valeurs de notre énumération LinuxFileAccess, nous allons utiliser les opérateurs de bits AND « & » OR « | » et XOR « ^ ». Le tableau suivant va vous permettre de réviser vos connaissances en algèbre de Boole :

| A | B | AND<br>A & B | OR<br>A \| B | XOR<br>A ^ B |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 |

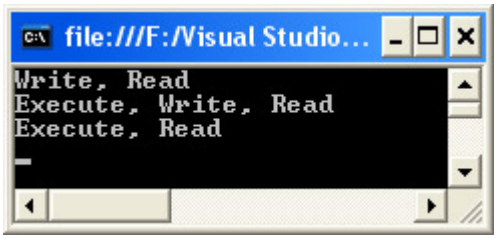
### III-B-1 - Associer ou dissocier des constantes

Pour combiner, ou au contraire retrancher des constantes dans une même variable, on utilisera les opérateurs de bits OR « | » et XOR « ^ ». Pour faire simple, en algèbre de Boole ce sont respectivement les équivalents de l'addition et de la soustraction :

```
LinuxFileAccess fileAccess = LinuxFileAccess.Read | LinuxFileAccess.Write;
Console.WriteLine(fileAccess);

// Équivalent à :
// fileAccess = fileAccess | LinuxFileAccess.Execute;
fileAccess |= LinuxFileAccess.Execute;
Console.WriteLine(fileAccess);

// Équivalent à :
// fileAccess = fileAccess ^ LinuxFileAccess.Write;
fileAccess ^= LinuxFileAccess.Write;
Console.WriteLine(fileAccess);
```




Ici nous initialisons tout d'abord notre variable avec les constantes Read et Write, puis nous y ajoutons la constante Execute, avant de finalement retrancher la constante Write. L'affichage dans la console atteste de ces manipulations successives.

Remarquez que lorsque l'on appelle la méthode ToString (Console.WriteLine le fait automatiquement) sur une variable combinant plusieurs constantes nommées, les différentes valeurs sont rassemblées en étant séparées par une virgule (qui n'est pas personnalisable) dans un ordre totalement arbitraire.

### III-B-2 - Tester si une constante est présente

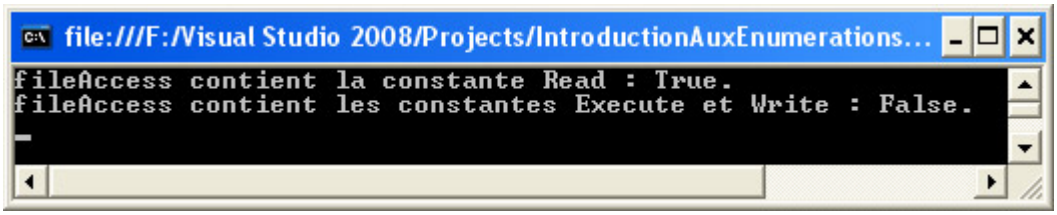
Toutes ces manipulations ne serviraient à rien si nous n'étions pas en mesure de tester si une variable contient l'une ou l'autre des constantes nommées. Mais la présence de l'attribut Flags va nous permettre d'aller un peu plus loin qu'avec une énumération classique : nous allons pouvoir tester aussi bien la présence d'une constante particulière que la présence d'une combinaison de plusieurs constantes, et cela indépendamment d'autres constantes qui pourraient aussi être définies dans le champ de bits.

La bonne nouvelle, c'est que le Framework .NET 4.0 a introduit dans la classe Enum une nouvelle méthode statique qui va grandement nous simplifier la tâche : HasFlag. Il s'agit d'une méthode d'extension que vous pouvez par conséquent exécuter directement contre votre variable :

```
LinuxFileAccess fileAccess = LinuxFileAccess.Read | LinuxFileAccess.Execute;

bool result1 = fileAccess.HasFlag(LinuxFileAccess.Read);
Console.WriteLine("fileAccess contient la constante Read : {0}.", result1);

bool result2 = fileAccess.HasFlag(LinuxFileAccess.Execute | LinuxFileAccess.Write);
Console.WriteLine("fileAccess contient les constantes Execute et Write : {0}.", result2);
```




Ces deux exemples illustrent bien que nous sommes en mesure de tester la présence d'une seule ou de plusieurs constantes combinées. Ici fileAccess contient bien la constante Read, mais pas à la fois les constantes Execute et Write (seulement Execute en l'occurrence).

Malheureusement si vous en êtes resté à la version 3.5 du framework .NET, il vous faudra recourir à l'opérateur de bits AND « & » :

```
LinuxFileAccess fileAccess = LinuxFileAccess.Read | LinuxFileAccess.Execute;
LinuxFileAccess combinationToTest, intersection;

// 1. Déclaration de la constante ou de la combinaison de constantes
// dont nous souhaitons vérifier l'existence dans la variable fileAccess
combinationToTest = LinuxFileAccess.Write;

// 2. Réduction de la variable fileAccess à cette ou ces constantes
// Ici intersection contiendra la constante fantôme de valeur zéro
intersection = fileAccess & combinationToTest;

// 3. Comparaison proprement dite
bool result1 = intersection == combinationToTest;
Console.WriteLine("fileAccess contient la constante Write : {0}.", result1);

/* ------------------------------------------------------------------- */

// 1. Déclaration de la constante ou de la combinaison de constantes
// dont nous souhaitons vérifier l'existence dans la variable fileAccess
combinationToTest = LinuxFileAccess.Read | LinuxFileAccess.Execute;

// 2. Réduction de la variable fileAccess à cette ou ces constantes
// Ici intersection contiendra bien la même combinaison que combinationToTest
intersection = fileAccess & combinationToTest;

// 3. Comparaison proprement dite
bool result2 = intersection == combinationToTest;
Console.WriteLine("fileAccess contient les constantes Read et Execute : {0}.", result2);
```

Nous avons deux groupes : celui correspondant aux constantes de la variable que nous souhaitons tester, et celui correspondant à la ou les constantes que nous recherchons. En appliquant l'opérateur AND entre ces deux groupes nous obtenons leur intersection, c'est-à-dire l'ensemble des constantes qu'ils ont en commun. S'ils n'en ont aucune, le résultat sera la constante par défaut de valeur 0, que celle-ci soit définie par l'énumération ou non.

Il ne reste donc ensuite qu'à comparer ce résultat avec le groupe des constantes recherchées afin de vérifier si celles-ci étaient bien un sous-ensemble de notre variable de départ, et correspondent donc bien à l'intersection.

Ce code peut paraître particulièrement verbeux, mais il ne faut pas oublier qu'il a été volontairement découpé en plusieurs étapes afin d'être plus clair. On peut facilement réduire le test à une seule ligne :

```
if ((fileAccess & LinuxFileAccess.Read) == LinuxFileAccess.Read)
    Console.WriteLine("fileAccess contient bien la constante Read.");
```

### III-B-3 - Simplifier les manipulations en proposant les valeurs intermédiaires

Un dernier conseil enfin : si vous souhaitez simplifier les manipulations avec une énumération décorée par l'attribut Flags, vous pouvez tout à fait ajouter à celle-ci des constantes nommées représentants les différentes valeurs intermédiaires possibles :

```
[Flags]
public enum LinuxFileAccess
{
    None = 0,
    Execute = 1,
    Write = 2,
    ExecuteAndWrite = Execute | Write, // = 1 + 2
    Read = 4,
    ExecuteAndRead = Execute | Read,   // = 1 + 4
    WriteAndRead = Write | Read,       // = 2 + 4
    All = Execute | Write | Read       // = 1 + 2 + 4
}
```

Notez qu'au passage nous utilisons l'astuce qui consiste à donner à une constante la valeur d'une autre constante. En utilisant l'opérateur de bits OR nous sommes en mesure de combiner directement ces constantes dans la déclaration de l'énumération.

Il est même possible - et c'est ici parfaitement justifié - de créer une constante nommée de valeur zéro, qui représentera le cas où aucune des constantes n'est sélectionnée. Ici par exemple, elle représente le cas où l'utilisateur n'a aucun droit d'accès sur le fichier.

## IV - Conclusion

Nous avons vu que les énumérations dans le Framework .NET permettent de faire circuler facilement des indications dans nos programmes, et que leur apparente simplicité cachait quelques mécanismes fort pratiques, mais aussi quelques pièges vicieux évitables en prenant quelques précautions.

Ainsi, pour se préserver des valeurs fantômes, on devrait toujours contrôler qu'une variable contient bien une valeur définie par l'énumération qui constitue son type.

Il est aussi préférable de toujours prendre en compte un cas par défaut dans les embranchements à base de switch et de if/else, quitte à ce que celui-ci lève une exception.

Enfin, parmi les bonnes pratiques on peut citer le fait d'avoir toujours une constante nommée associée à la valeur par défaut zéro, sauf cas bien particuliers.

## IV-A - Pour aller plus loin

Les énumérations sur MSDN :

- **enum** ;
- **L'attribut Flags** ;
- **Les types d'énumérations** (classique ou avec bits indicateurs) ;
- **La classe Enum** (en anglais, plus complet que le version traduite) ;
- **Les méthodes statiques de la classe Enum**.

## IV-B - Remerciements

Merci à **Claude Leloup** pour ses corrections et sa relecture attentive, et à **JolyLoic** pour sa suggestion sur les nouveautés du Framework .NET 4.0.