# Associating enums with strings in C#

Asked 10 years, 6 months ago Active yesterday Viewed 268k times

285

91

I know the following is not possible because the Enumeration's type has to be an int

```
enum GroupTypes
{
    TheGroup = "OEM",
    TheOtherGroup = "CMB"
}
```

From my database I get a field with incomprehensive codes (the `OEM` and `CMB` s). I would want to make this field into an `enum` or something else understandable. Because if the target is readability, the solution should be terse.

What other options do I have?

c# .net

edited Sep 19 at 3:15
Chad
967 1 14 37

asked Mar 10 '09 at 15:33

Boris Callens
38.4k 69 199 292

possible duplicate of Enum ToString – nawfal Jun 8 '13 at 22:41

9 I'm not sure why most of the answers don't just use "const string" and instead they're making custom classes. – CTS_AE May 15 '15 at 22:23

You may not be able to use strings, but you can use chars just fine. That's an option if you can use single-letter values. – T. Sar Jun 14 '17 at 15:30

1 Genuinely confused as to why the solution proposed above by CTS_AE is not even in the top three answers. – Sinjai Sep 5 '17 at 21:57

@Sinjai Explicit grouping of related values will outweigh the penalty of an imperceptible performance loss, especially in an API or reusable component. – person27 Aug 26 '18 at 4:18

## 30 Answers

320

I like to use **properties in a class** instead of methods, since they look more enum-like.

Here's a example for a Logger:

```
public class LogCategory
{
 private LogCategory(string value) { Value = value; }

 public string Value { get; set; }

 public static LogCategory Trace { get { return new LogCategory("Trace"); } }
 public static LogCategory Debug { get { return new LogCategory("Debug"); } }
 public static LogCategory Info { get { return new LogCategory("Info"); } }
 public static LogCategory Warning { get { return new LogCategory("Warning"); } }
 public static LogCategory Error { get { return new LogCategory("Error"); } }
}
```

Pass in **type-safe string values** as a parameter:

```
public static void Write(string message, LogCategory logCategory)
{
   var log = new LogEntry { Message = message };
   Logger.Write(log, logCategory.Value);
}
```

Usage:

```
Logger.Write("This is almost like an enum.", LogCategory.Info);
```

answered Aug 27 '09 at 20:12

Even Mien
**22.5k** 37 105 116

4 Only down side I can come up with is that it would be a tiny bit slower, but this would in most cases be neglectable. And it wouldn't have the exact same behaviour in the editor. E.G.: switching over this one wouldn't automatically fill in a case for each possibility. Other than those minor points, I think this is probably a rather simple solution. – Boris Callens Aug 28 '09 at 7:06

3 And it's easy to use Dictionary<LogCategory, Action/Func> as a switch. :) – Arnis Lapsa Dec 5 '09 at 16:06

12 For my own use, I expanded upon this concept, overriding the `ToString` method to return `Value` . And then provided implicit cast operators to and from a string. `public static implicit operator String(LogCategory category) { return Value; }` . – Zarepheth Jan 24 '14 at 20:17

5 What about using this in switch cases? – David Feb 3 '15 at 10:56

148

You could also use the extension model:

```
public enum MyEnum
{
    [Description("String 1")]
    V1= 1,
    [Description("String 2")]
    V2= 2
}
```

Your Extension Class

```
public static class MyEnumExtensions
{
    public static string ToDescriptionString(this MyEnum val)
    {
        DescriptionAttribute[] attributes = (DescriptionAttribute[])val
           .GetType()
           .GetField(val.ToString())
           .GetCustomAttributes(typeof(DescriptionAttribute), false);
        return attributes.Length > 0 ? attributes[0].Description : string.Empty;
    }
}
```

usage:

```
MyEnum myLocal = MyEnum.V1;
print(myLocal.ToDescriptionString());
```

edited Sep 24 '18 at 23:25

kmote
**11.9k** 8 53 78

answered Mar 10 '09 at 15:53

Glennular
**14.4k** 8 50 73

4 I can't help thinking that reflecting the enum every time you want do display the text sounds kind of painful from a performance perspective! – Liath Sep 1 '14 at 15:03

3 @Liath - The ` .ToString() ` already uses reflection, so you're not really losing anything with this approach, and gaining readibility – James King Mar 28 '16 at 18:24

1 Could you make the Extension generic so it applied automatically to all Enums? – erosebe May 25 '17 at 21:06

1 To make generic, use `public static string ToDescriptionString(this Enum ...` i.e. without explicitly typing to `MyEnum` . – LeeCambl Sep 15 '18 at 13:29

79

How about using a static class with constants? The client code will look no different from enums.

```
static class GroupTypes
{
  public const string TheGroup = "OEM";
  public const string TheOtherGroup = "CMB";
}

void DoSomething(GroupTypes groupType)
{
  if(groupType == GroupTypes.TheOtherGroup)
  {
    //Launch nuclear bomb
  }
}
```

edited Jun 9 '16 at 1:17

answered Apr 15 '11 at 9:16

rpattabi

**6,924** 2 35 43

9 Agreed. I'm having trouble seeing the purpose behind the more complex solutions, except maybe to be able to switch over the resulting "enum". – fakeleft Dec 16 '11 at 12:55

@fakeleft you cannot use a static class type with a generic type (template), and maybe other limitations, I think that is why people prefer the "more complex" solutions. – eselk Nov 24 '15 at 16:49

2 The constants need to be internal or public for this to work though – arviman Nov 30 '15 at 7:05

'18 at 11:40

28

You can add attributes to the items in the enumeration and then use reflection to get the values from the attributes.

You would have to use the "field" specifier to apply the attributes, like so:

```
enum GroupTypes
{
    [field:Description("OEM")]
    TheGroup,

    [field:Description("CMB")]
    TheOtherGroup
}
```

You would then reflect on the static fields of the type of the enum (in this case GroupTypes) and get the `DescriptionAttribute` for the value you were looking for using reflection:

```
public static DescriptionAttribute GetEnumDescriptionAttribute<T>(
    this T value) where T : struct
{
    // The type of the enum, it will be reused.
    Type type = typeof(T);

    // If T is not an enum, get out.
    if (!type.IsEnum)
        throw new InvalidOperationException(
            "The type parameter T must be an enum type.");

    // If the value isn't defined throw an exception.
    if (!Enum.IsDefined(type, value))
        throw new InvalidEnumArgumentException(
            "value", Convert.ToInt32(value), type);

    // Get the static field for the value.
    FieldInfo fi = type.GetField(value.ToString(),
        BindingFlags.Static | BindingFlags.Public);

    // Get the description attribute, if there is one.
    return fi.GetCustomAttributes(typeof(DescriptionAttribute), true)
```

I opted to return the `DescriptionAttribute` itself above, in the event that you want to be able to determine whether or not the attribute is even applied.

edited Sep 11 '12 at 18:56

answered Mar 10 '09 at 15:37

casperOne

**66.9k** 14 161 228

Although I will remember this for more complex situations, it is rather complex for a situation with the complexity level of what I stated in the OP – Boris Callens Sep 24 '12 at 10:15

22

You can do it very easily actually. Use the following code.

```
enum GroupTypes
{
   OEM,
   CMB
};
```

Then when you want to get the string value of each enum element just use the following line of code.

```
String oemString = Enum.GetName(typeof(GroupTypes), GroupTypes.OEM);
```

I've used this method successfully in the past, and I've also used a constants class to hold string constants, both work out pretty well, but I tend to prefer this.

answered Dec 5 '09 at 15:50

Arthur C

**374** 2 5

I was thinking the same thing, but there must be some catch to this... Otherwise I would suspect more people would suggest this (Maybe I'm just paranoid). – Matthijs Wessels Jan 11 '10 at 14:14

The only catch I'm aware of to this is that I believe it uses reflection to figure out the string. As a result if I'm just after a solution to keep track of a constant string, then I typically will use a class to store the majority of my constant strings. However if I have situation where an Enum is the right solution (regardless of getting a descriptive string about my Enum elements), then rather than have an extra string floating around somewhere to

spaces in the text (more details here). – SharpC Aug 7 '13 at 19:15

11 No, this is just getting an enum value's name, not assigning a string to an enum value. The goal of the OP is to have a string different from the enum value eg : TheGroup = "OEM", TheOtherGroup = "CMB". – Tim Autin Nov 3 '14 at 10:29

3 I agree with @Tim's comment, this is *not* what the OP is trying to do. If you're wondering what a use case of this is, consider a situation where a device takes strings as commands, but there needs to be a "human readable" version of the command as well. I had need of this to associate something like "Update Firmware" with the command "UPDATEFW". – JYelton Jan 2 '15 at 20:19

---

13

Create a second enum, for your DB containing the following:

```
enum DBGroupTypes
{
    OEM = 0,
    CMB = 1
}
```

Now, you can use Enum.Parse to retrieve the correct DBGroupTypes value from the strings "OEM" and "CMB". You can then convert those to int and retrieve the correct values from the right enumeration you want to use further in your model.

edited Mar 21 '10 at 10:31

abatishchev
**72.3k** 70 269 405

answered Mar 10 '09 at 15:36

Dave Van den Eynde
**12.9k** 6 55 78

This seems to be an extra step in the process, why not one class that handles everything? – C. Ross Mar 10 '09 at 19:02

11 As opposed to using attributes and reflection? – Dave Van den Eynde Mar 16 '09 at 8:08

---

13

Use a class.

**Edit: Better example**

```
class StarshipType
{
    private string _Name;
```

```
    public static readonly StarshipType Light = new StarshipType("Light");
    public static readonly StarshipType Mediumweight = new StarshipType("Mediumweight");
    public static readonly StarshipType Heavy = new StarshipType("Heavy");
    public static readonly StarshipType Superheavy = new StarshipType("Superheavy");

    public string Name
    {
        get { return _Name; }
        private set { _Name = value; }
    }

    public static IList<StarshipType> StarshipTypes
    {
        get { return _StarshipTypes; }
    }

    private StarshipType(string name, int systemRatio)
    {
        Name = name;
        _StarshipTypes.Add(this);
    }

    public static StarshipType Parse(string toParse)
    {
        foreach (StarshipType s in StarshipTypes)
        {
            if (toParse == s.Name)
                return s;
        }
        throw new FormatException("Could not parse string.");
    }
}
```

edited Aug 15 '17 at 16:30

Kevin Doyon
**2,403** 24 34

answered Mar 10 '09 at 15:38

C. Ross
**18.9k** 35 133 223

1 Difficult to go from the code back to the descriptive name. You would have to use reflection over all the const fields to search for a match. – andleer Mar 10 '09 at 15:42

1 I see your point. I will upload a version that acutally works later, but I admit it's pretty heavy. – C. Ross Mar 11 '09 at 13:28

My version based on C. Ross's solution stackoverflow.com/a/48441114/3862615 – Roman M Jan 25 '18 at 10:54

Try adding constants to a static class. You don't end up with a Type, but you will have readable, organised constants:

11

```
public static class GroupTypes {

    public const string TheGroup = "OEM";
    public const string TheOtherGroup = "CMB";

}
```

edited Sep 18 at 21:59
Chad
967 1 14 37

answered Mar 10 '09 at 15:38
darasd
2,144 2 17 36

1 Difficult to go from the code back to the descriptive name. You would have to use reflection over all the const fields to search for a match. – andleer Mar 10 '09 at 15:43

1 @andleer I don't understand your concern. This is the solution I use. – VSO May 15 '18 at 19:37

Yeah this is actually precisely what I wanted. And this is the most concise/elegant solution I see, just as if I were defining an enumeration w/ int values - but with string values instead. 100% perfect. – Chad Sep 18 at 20:08

Here is the extension method that I used to get the enum value as string. First here is the enum.

6

```
public enum DatabaseEnvironment
{
    [Description("AzamSharpBlogDevDatabase")]
    Development = 1,
    [Description("AzamSharpBlogQADatabase")]
    QualityAssurance = 2,
    [Description("AzamSharpBlogTestDatabase")]
    Test = 3
}
```

The Description attribute came from System.ComponentModel.

And here is my extension method:

```
    // get the field
    var field = environment.GetType().GetField(environment.ToString());
    var customAttributes = field.GetCustomAttributes(typeof (DescriptionAttribute),
false);

    if(customAttributes.Length > 0)
    {
        return (customAttributes[0] as DescriptionAttribute).Description;
    }
    else
    {
        return environment.ToString();
    }
}
```

Now, you can access the enum as string value using the following code:

```
[TestFixture]
public class when_getting_value_of_enum
{
    [Test]
    public void should_get_the_value_as_string()
    {

Assert.AreEqual("AzamSharpBlogTestDatabase",DatabaseEnvironment.Test.GetValueAsString());

    }
}
```

edited Sep 1 '14 at 13:26

Shoaib Shakeel
**1,101** 13 22

answered Mar 15 '10 at 22:38

azamsharp
**11.4k** 33 130 216

---

6

Another way to deal with the problem, is to have a enum and a array of strings that will map the enum values with the list of strings:

```
public enum GroupTypes
{
    TheGroup  = 0,
```

```
string[] GroupTypesStr = {
    "OEM",
    "CMB"
};
```

you may use it something like this:

```
Log.Write(GroupTypesStr[(int)GroupTypes.TheOtherGroup]);
```

It will prompt CMB

PROS:

1. Easy and clean code.
2. High Performance (specially in comparison with those approaches that uses classes)

CONS:

1. Prone to mess up the list when editing it, but it will be okay for a short list.

edited Jan 4 '18 at 3:05

answered Dec 29 '17 at 20:16

Luis Orantes

**105** 1 9

---

4

Have you considered a lookup table using a Dictionary?

```
enum GroupTypes
{
    TheGroup,
    TheOtherGroup
}

Dictionary<string, GroupTypes> GroupTypeLookup = new Dictionary<string, GroupTypes>();
// initialize lookup table:
GroupTypeLookup.Add("OEM", TheGroup);
GroupTypeLookup.Add("CMB", TheOtherGroup);
```

answered Mar 10 '09 at 15:39

Jim Mischel
**110k** 12 142 265

How do you easily get the key for a given value? – eglasius Mar 10 '09 at 15:43

The question didn't ask to go the other way. But it'd be simple enough to build another dictionary that goes the other way. That is, Dictionary<GroupTypes, string>. – Jim Mischel Mar 11 '09 at 15:37

3

C# doesn't support enumerated strings, but for most situations you can use a List or Dictionary to get the desired effect.

E.g. To print pass/fail results:

```
List<string> PassFail = new List<string> { "FAIL", "PASS" };
bool result = true;
Console.WriteLine("Test1: " + PassFail[result.GetHashCode()]);
```

answered Mar 5 '14 at 17:39



user1441126
**129** 2

2

I would make it into a class an avoid an enum altogether. And then with the usage of a typehandler you could create the object when you grab it from the db.

IE:

```
public class Group
{
    public string Value{ get; set; }
    public Group( string value ){ Value = value; }
    public static Group TheGroup() { return new Group("OEM"); }
    public static Group OtherGroup() { return new Group("CMB"); }

}
```

Bryan Rowe
**7,304** 4 22 33

2

I would just create a dictionary and use the code as the key.

Edit: To address the comment about doing a reverse lookup (finding the key), this would not be terribly efficient. If this is necessary, I would write a new class to handle it.

edited Mar 10 '09 at 15:45

answered Mar 10 '09 at 15:36

jhale
**1,029** 2 11 20

Depends if it's really constant ... – C. Ross Mar 10 '09 at 15:39

Also can you easily grab a key for a given value? – eglasius Mar 10 '09 at 15:39

To C.Ross - I'm not sure what you mean. You can read the values in from a db and dynamically populate the dictionary. – jhale Mar 10 '09 at 15:44

2

My first question - Do you have access to the Database itself? This should be normalized in the database, ideally, otherwise, any solution is going to be prone to error. In my experience, data fields full of "OEM" and "CMB" tend to wind up having things like "oem " and other 'crap data' mixed in over time.... If you can normalize it, you could use the key in the table containing the elements as your Enum, and you're done, with a much cleaner structure.

If that's not available, I'd make your Enum, and make a class to parse your string into the Enum for you. This would at least give you some flexibility in handling non-standard entries and much more flexibility for trapping or handling errors than doing any of the workarounds using Enum.Parse/Reflection/etc. A dictionary would work, but could break down if you ever have case issues, etc.

I'd recommend writing a class so you can do:

```
// I renamed this to GroupType, since it sounds like each element has a single type...
GroupType theType = GroupTypeParser.GetGroupType(theDBString);
```

This preserves most of your readability without having to change the DB.


**483k** 60 1013 1297

2

If I understand correctly, you need a conversion from string to enum:

```
enum GroupTypes {
    Unknown = 0,
    OEM = 1,
    CMB = 2
}
static GroupTypes StrToEnum(string str){
    GroupTypes g = GroupTypes.Unknown;
    try {
        object o = Enum.Parse(typeof(GroupTypes), str, true);
        g = (GroupTypes)(o ?? 0);
    } catch {
    }
    return g;
}
// then use it like this
GroupTypes g1 = StrToEnum("OEM");
GroupTypes g2 = StrToEnum("bad value");
```

You can make it more fancy with generics for the enum type if you wish.

edited Nov 24 '14 at 3:17


Moes
**9,871** 4 45 54

answered Nov 20 '14 at 19:33


dmihailescu
**1,292** 15 14

2

In VS 2015, you can use nameof

```
public class LogCategory
{
    public static string Trace;
    public static string Debug;
    public static string Info;
    public static string Warning;
    public static string Error;
```

Usage:

```
Logger.Write("This is almost like an enum.", nameof(LogCategory.Info));
```

answered Jan 11 '17 at 7:59

Claus Elmann
37 2

---

2

A small tweak to Glennular Extension method, so you could use the extension on other things than just ENUM's;

```
using System;
using System.ComponentModel;
namespace Extensions {
    public static class T_Extensions {
        /// <summary>
        /// Gets the Description Attribute Value
        /// </summary>
        /// <typeparam name="T">Entity Type</typeparam>
        /// <param name="val">Variable</param>
        /// <returns>The value of the Description Attribute or an Empty String</returns>
        public static string Description<T>(this T t) {
            DescriptionAttribute[] attributes =
(DescriptionAttribute[])t.GetType().GetField(t.ToString()).GetCustomAttributes(typeof(Desc
 false);
            return attributes.Length > 0 ? attributes[0].Description : string.Empty;
        }
    }
}
```

Or Using Linq

```
using System;
using System.ComponentModel;
using System.Linq;

namespace Extensions {
```

```
            ((DescriptionAttribute[])t
            ?.GetType()
            ?.GetField(t?.ToString())
            ?.GetCustomAttributes(typeof(DescriptionAttribute), false))
            ?.Select(a => a?.Description)
            ?.FirstOrDefault()
            ?? string.Empty;
    }
}
```

edited Oct 18 '17 at 15:30

answered Oct 18 '17 at 13:29

Mike

**559** 3 7

---

2

```
public class DataType
{
    private readonly string value;
    private static readonly Dictionary<string, DataType> predefinedValues;

    public static readonly DataType Json = new DataType("json");
    public static readonly DataType Xml = new DataType("xml");
    public static readonly DataType Text = new DataType("text");
    public static readonly DataType Html = new DataType("html");
    public static readonly DataType Binary = new DataType("binary");

    static DataType()
    {
        predefinedValues = new Dictionary<string, DataType>();
        predefinedValues.Add(Json.Value, Json);
        predefinedValues.Add(Xml.Value, Xml);
        predefinedValues.Add(Text.Value, Text);
        predefinedValues.Add(Html.Value, Html);
        predefinedValues.Add(Binary.Value, Binary);
    }

    private DataType(string value)
    {
        this.value = value;
    }

    public static DataType Parse(string value)
```

```
            if (string.IsNullOrEmpty(value))
                throw exception;

            string key = value.ToLower();
            if (!predefinedValues.ContainsKey(key))
                throw exception;

            return predefinedValues[key];
        }

        public string Value
        {
            get { return value; }
        }
    }
```

answered Jan 25 '18 at 10:53

Roman M

**159** 2 8

---

1

I even implemented a few enums as suggested by @Even (via `class X` and `public static X` members), just to find out later that these days, starting .Net 4.5, there's *the right* `ToString()` method.

Now I'm reimplementing everything back to enums.

answered Nov 19 '15 at 13:37

bohdan_trotsenko

**3,045** 32 57

---

1

This is a way to use it **as a strongly typed parameter or as a string** :

```
public class ClassLikeEnum
{
    public string Value
    {
        get;
```

```
    ClassLikeEnum(string value)
    {
        Value = value;
    }

    public static implicit operator string(ClassLikeEnum c)
    {
        return c.Value;
    }

    public static readonly ClassLikeEnum C1 = new ClassLikeEnum("RandomString1");
    public static readonly ClassLikeEnum C2 = new ClassLikeEnum("RandomString2");
}
```

answered Apr 25 '17 at 14:50

Nicolas Bodin-Ripert
**716** 9 17

---

1

You can use two enums. One for the database and the other for readability.

You just need to make sure they stay in sync, which seems like a small cost. You don't have to set the values, just set the positions the same, but setting the values makes it very clear the two enums are related and prevents errors from rearranging the enum members. And a comment lets the maintenance crew know these are related and must be kept in sync.

```
// keep in sync with GroupTypes
public enum GroupTypeCodes
{
    OEM,
    CMB
}

// keep in sync with GroupTypesCodes
public enum GroupTypes
{
    TheGroup = GroupTypeCodes.OEM,
    TheOtherGroup = GroupTypeCodes.CMB
}
```

To use it you just convert to the code first:

```
GroupTypes myGroupType = GroupTypes.TheGroup;
string valueToSaveIntoDatabase = ((GroupTypeCodes)myGroupType).ToString();
```

Then if you want to make it even more convenient you can add an extension function that only works for this type of enum:

```
public static string ToString(this GroupTypes source)
{
    return ((GroupTypeCodes)source).ToString();
}
```

and you can then just do:

```
GroupTypes myGroupType = GroupTypes.TheGroup;
string valueToSaveIntoDatabase = myGroupType.ToString();
```

edited May 17 '17 at 15:34

answered Aug 5 '16 at 21:15

toddmo

**10.8k** 8 69 80

That is a bad practice: With a dependent `enum` an intended value change in one might uninentionally mess up the other. – Lorenz Lo Sauer Sep 3 '16 at 13:40

1

I was basically looking for the Reflection answer by @ArthurC

Just to extend his answer a little bit, you can make it even better by having a generic function:

```
// If you want for a specific Enum
private static string EnumStringValue(GroupTypes e)
{
    return EnumStringValue<GroupTypes>(e);
}

// Generic
private static string EnumStringValue<T>(T enumInstance)
{
```

Then you can just wrap whatever you have

```
EnumStringValue(GroupTypes.TheGroup) // if you incorporate the top part
```

or

```
EnumStringValue<GroupTypes>(GroupTypes.TheGroup) // if you just use the generic
```

answered Jan 18 '18 at 21:54

ThinkBonobo

**8,129** 5 41 60

---

0

Based in other opinions, this is what I come up with. This approach avoids having to type .Value where you want to get the constant value.

I have a base class for all string enums like this:

```
using System;
using Newtonsoft.Json;

[JsonConverter(typeof(ConstantConverter))]
public class StringEnum: IConvertible
{
    public string Value { get; set; }

    protected StringEnum(string value)
    {
        Value = value;
    }

    public static implicit operator string(StringEnum c)
    {
        return c.Value;
    }
    public string ToString(IFormatProvider provider)
    {
        return Value;
    }
```

```
        {
            throw new NotImplementedException();
        }

        public bool ToBoolean(IFormatProvider provider)
        {
            throw new NotImplementedException();
        }
        //The same for all the rest of IConvertible methods
    }
```

The JsonConverter is like this:

```
using System;
using Newtonsoft.Json;

class ConstantConverter : JsonConverter
{
    public override bool CanConvert(Type objectType)
    {
        return true;
    }

    public override object ReadJson(JsonReader reader, Type objectType, object
existingValue, JsonSerializer serializer)
    {
        throw new NotImplementedException();
    }

    public override void WriteJson(JsonWriter writer, object value, JsonSerializer
serializer)
    {
        if (value == null)
        {
            serializer.Serialize(writer, null);
        }
        else
        {
            serializer.Serialize(writer, value.ToString());
        }
    }
}
```

And an actual string enum will be something like this:

```
public sealed class Colors : StringEnum
{
    public static Colors Red { get { return new Catalog("Red"); } }
    public static Colors Yellow { get { return new Catalog("Yellow"); } }
    public static Colors White { get { return new Catalog("White"); } }

    private Colors(string value) : base(value) { }
}
```

And with this, you can just use Color.Red to even serialize to json without using the Value property

answered Feb 8 '15 at 20:54

ayepiz
**20** 3

---

0

I didn't need anything robust like storing the string in attributes. I just needed to turn something like `MyEnum.BillEveryWeek` into "bill every week" or `MyEnum.UseLegacySystem` into "use legacy system"--basically split the enum by its camel-casing into individiual lower-case words.

```
public static string UnCamelCase(this Enum input, string delimiter = " ", bool
preserveCasing = false)
{
    var characters = input.ToString().Select((x, i) =>
    {

       if (i > 0 && char.IsUpper(x))
       {
           return delimiter + x.ToString(CultureInfo.InvariantCulture);
       }
       return x.ToString(CultureInfo.InvariantCulture);

    });

    var result = preserveCasing
       ? string.Concat(characters)
       : string.Concat(characters).ToLower();

    var lastComma = result.LastIndexOf(", ", StringComparison.Ordinal);

    if (lastComma > -1)
```

```
    return result;
}
```

`MyEnum.UseLegacySystem.UnCamelCase()` outputs "use legacy system"

If you have multiple flags set, it will turn that into plain english (comma-delimited except an "and" in place of the last comma).

```
var myCustomerBehaviour = MyEnum.BillEveryWeek | MyEnum.UseLegacySystem |
MyEnum.ChargeTaxes;

Console.WriteLine(myCustomerBehaviour.UnCamelCase());
//outputs "bill every week, use legacy system and charge taxes"
```

answered Jul 20 '16 at 2:34

Chad Hedgcock
**7,914** 3 24 36

---

0

I've done something like this;

```
public enum BusinessUnits
{
    NEW_EQUIPMENT = 0,
    USED_EQUIPMENT = 1,
    RENTAL_EQUIPMENT = 2,
    PARTS = 3,
    SERVICE = 4,
    OPERATOR_TRAINING = 5
}

public class BusinessUnitService
{
    public static string StringBusinessUnits(BusinessUnits BU)
    {
        switch (BU)
        {
            case BusinessUnits.NEW_EQUIPMENT: return "NEW EQUIPMENT";
            case BusinessUnits.USED_EQUIPMENT: return "USED EQUIPMENT";
            case BusinessUnits.RENTAL_EQUIPMENT: return "RENTAL EQUIPMENT":
```

```
            default: return String.Empty;
        }
    }
}
```

Call it with this;

```
BusinessUnitService.StringBusinessUnits(BusinessUnits.PARTS)
```

answered Jul 6 '18 at 21:31

Mahib
**2,545** 3 40 52

---

0

I wanted to avoid using string literals completely, and also I didn't need to have space in item descriptions. More importantly, I wanted to have a mechanism to check if the provided string is a valid item, so I came up with this solution:

```
public class Seasons
{
    public static string Spring { get; }
    public static string Summer { get; }
    public static string Fall { get; }
    public static string Winter { get; }

    public static bool IsValid(string propertyName)
    {
        if (string.IsNullOrEmpty(propertyName))
        {
            return false;
        }

        try
        {
            return typeof(Seasons).GetProperty(propertyName) != null;
        }
        catch
        {
            return false;
        }
    }
```

And here is how it works:

```
void Main()
{
    string s = nameof(Seasons.Fall);
    Console.WriteLine($"Fall is valid: {Seasons.IsValid(s)}"); // true

    s = "WrongSeason";
    Console.WriteLine($"WrongSeason is valid: {Seasons.IsValid(s)}"); // false
}
```

I tried to refactor IsValid() into a base class and use reflection to read the type (MethodBase.GetCurrentMethod().DeclaringType), but since I wanted to have it static, it returns the base class type, not the inherited type. Your remedy to this will be very welcomed! Here is what I was trying to achieve:

```
public  class Seasons : ConstantStringsBase
{
    // ... same
}

public  class ConstantStringsBase
{
    public static bool IsValid(string propertyName)
    {
        return MethodBase.GetCurrentMethod().DeclaringType.GetProperty(propertyName) !=
null;
    }
}
```

answered Mar 8 at 17:46

Siavash Mortazavi
**161** 3 10

---

0

Following the answer of @Even Mien I have tried to go a bit further and make it Generic, I seem to be almost there but one case still resist and I probably can simplify my code a bit.
I post it here if anyone see how I could improve and especially make it works as I can't assign it from a string

So Far I have the following results:

```
        Console.WriteLine(TestEnum.Test1);//displays "TEST1"

        bool test = "TEST1" == TestEnum.Test1; //true

        var test2 = TestEnum.Test1; //is TestEnum and has value

        string test3 = TestEnum.Test1; //test3 = "TEST1"

        var test4 = TestEnum.Test1 == TestEnum.Test2; //false
         EnumType<TestEnum> test5 = "TEST1"; //works fine

        //TestEnum test5 = "string"; DOESN'T compile .... :(:(
```

Where the magics happens :

```
public abstract  class EnumType<T>  where T : EnumType<T>
{

    public  string Value { get; set; }

    protected EnumType(string value)
    {
        Value = value;
    }


    public static implicit operator EnumType<T>(string s)
    {
        if (All.Any(dt => dt.Value == s))
        {
            Type t = typeof(T);

            ConstructorInfo ci = t.GetConstructor(BindingFlags.Instance |
BindingFlags.NonPublic,null, new Type[] { typeof(string) }, null);

            return (T)ci.Invoke(new object[] {s});
        }
        else
        {
            return null;
        }
    }
```

```
    }


    public static bool operator ==(EnumType<T> ct1, EnumType<T> ct2)
    {
        return (string)ct1 == (string)ct2;
    }

    public static bool operator !=(EnumType<T> ct1, EnumType<T> ct2)
    {
        return !(ct1 == ct2);
    }


    public override bool Equals(object obj)
    {
        try
        {
            return (string)obj == Value;
        }
        catch
        {
            return false;
        }
    }

    public override int GetHashCode()
    {
        return Value.GetHashCode();
    }

    public static IEnumerable<T> All
     => typeof(T).GetProperties()
       .Where(p => p.PropertyType == typeof(T))
       .Select(x => (T)x.GetValue(null, null));



}
```

I only then have to declare this for my enums:

```
public class TestEnum : EnumType<TestEnum>
{
```

```
    public static TestEnum Test1 { get { return new TestEnum("TEST1"); } }
    public static TestEnum Test2 { get { return new TestEnum("TEST2"); } }
}
```

edited Jun 7 at 14:41

answered Jun 6 at 17:26

Lomithrani
**1,065** 1 13 19

---

0

Taken from @EvenMien and added in some of the comments. (Also for my own use case)

```
public struct AgentAction
{
    private AgentAction(string value) { Value = value; }

    public string Value { get; private set; }

    public override string ToString()
    {
        return this.Value;
    }

    public static AgentAction Login = new AgentAction("Logout");
    public static AgentAction Logout = new AgentAction("Logout");

    public static implicit operator string(AgentAction action) { return
action.ToString(); }
}
```

edited Jul 20 at 5:20

answered Jul 19 at 22:55

moldypenguins
**170** 1 7

---

0

```
public class DatabasePreference {
    public DatabasePreference([CallerMemberName] string preferenceName = "") {
        PreferenceName = preferenceName;
```

```
//Declare names here
public static DatabasePreference ScannerDefaultFlashLight = new DatabasePreference();
public static DatabasePreference ScannerQrCodes = new DatabasePreference();
public static DatabasePreference Scanner1dCodes = new DatabasePreference();
```

This work is using `CallerMemberName` to minimize the coding

```
Console.WriteLine(ScannerDefaultFlashLight.PreferenceName);
Console.WriteLine(ScannerDefaultFlashLight.Scanner1dCodes);
```

output:

```
ScannerDefaultFlashLight
Scanner1dCodes
```

edited yesterday

answered yesterday

nyconing

**603** 7 24