



Number.cs for General Purpose Number Handling in C#



GerVenson

17 Sep 2020 Ms-RL

An wrapper class for Number operations in C#

As C# does not offer a underlying base class for Numbers, you have to write one that can work with all numbers without specifying the Type directly

[Download Number_clean_v2.zip - 5 KB](#)

[Download Number_Performance_Optimised_.zip - 4.6 KB](#)

[Download Number.zip - 3.8 KB](#)

<https://github.com/JPVenson/morestachio/blob/master/Morestachio/Helper/Number.cs>

Introduction

As I am working on my [Project Morestachio](#), I wanted to allow users to either supply numbers from code but also allow number operations in my Template. That was where the trouble started and I needed to operate on two numbers that might not be the same or have unknown types. As I already had to write a parser to get the number from the text template, I decided to allow all numbers to be wrapped into the *Number.cs* class and implemented all common operations on it.

Hint: The source on github contains references to the **Morestachio** Project within the **MorestachioFormatter** region. If you want to use only the *Number.cs* class, you are safe to delete this whole region. The *Number.cs* attached to this tip does not contain this region!

Background

Number.cs is of interest for everyone who got numbers but without knowing what the number represents. *Number.cs* can parse any number from **string** according the rules of C# ([Integral numeric types \(C# reference\)](#) and [Real literals](#)).

Using the Code

Number.cs has three different ways in which to create a new instance of *Number.cs*:

1. `bool Number.TryParse(string, CultureInfo, out Number).`

This tries to parse a **string** into a number based on the literal and the C# rules.

2. `Number.ctor(long|ulong|int|uint|byte|sbyte|short|ushort|float|double|decimal)`

Use the constructor to create a new number with the internal .NET type

3. implicit operator

`Number(Long|ulong|int|uint|byte|sbyte|short|ushort|float|double|decimal)`

Number.cs implements the implicit casting operator for all .NET Number types

You can get a list of all supported .NET types by calling `Number.CsFrameworkIntegralTypes` and `Number.CsFrameworkFloatingPointNumberTypes` respectively. Note that those lists are only informational.

Now that you got your instance of *Number.cs*, you can make operations with it. There are two general ways in operating with *Number.cs*:

1. Call the operation functions like:

- ⊗ `Add(Number): Number`
- ⊗ `Substract(Number): Number`
- ⊗ `Multiply(Number): Number`
- ⊗ `Divide(Number): Number`
- ⊗ `Modulo(Number): Number`
- ⊗ `ShiftLeft(Number): Number`
- ⊗ `ShiftRight(Number): Number`
- ⊗ `GreaterThan(Number): bool`
- ⊗ `SmallerThan(Number): bool`
- ⊗ `Equals(Number): bool`
- ⊗ `Same(Number): bool`
- ⊗ `IsNaN(): bool`
- ⊗ `Max(Number): Number`
- ⊗ `Min(Number): Number`
- ⊗ `Pow(Number): Number`
- ⊗ `Log(Number): Number`
- ⊗ `Log10(Number): Number`
- ⊗ `Negate(): Number`
- ⊗ `Abs(): Number`
- ⊗ `Round(Number): Number`

Note: `Equals(Number)` will first check if both numbers are of the same .NET type and then call `Same(Number)` for value equality check

2. Use C# operators. *Number.cs* implements the following operators on itself:

- `Number + Number`
- `Number++`
- `Number - Number`
- `Number--`
- `Number << Number`
- `Number >> Number`
- `Number == Number`
- `Number != Number`
- `Number < Number`
- `Number > Number`
- `Number <= Number`
- `Number >= Number`

In addition to these operations on itself (`Number + Number` or `Number < Number`), you can also use the following operators on all other .NET number types:

- `Number + long|ulong|int|uint|byte|sbyte|short|ushort|float|double|decimal`
- `Number - long|ulong|int|uint|byte|sbyte|short|ushort|float|double|decimal`

- `Number * long|ulong|int|uint|byte|sbyte|short|ushort|float|double|decimal`
- `Number / long|ulong|int|uint|byte|sbyte|short|ushort|float|double|decimal`
- `Number % long|ulong|int|uint|byte|sbyte|short|ushort|float|double|decimal`

These operators just call the methods from point 1.

`Number.cs` takes care of the implicit conversion of values from an operation. That means like in cs if you add two numbers together where one number is a floating point number, the result will always be a floating point number of the same type and if you add two integers together, the result will always be of the type that has more precision. This is done by a rating of numbers that exists within the `Number.CsFrameworkFloatingPointNumberTypes` and `Number.CsFrameworkIntegralTypes`. The function `Number.GetOperationTargetType` will return the type that takes prevalence over another. The operation then always looks the same like this:

Example Add Method:

```

/// <summary>
/// Adds the two numbers together
/// </summary>
/// <param name="other"></param>
/// <returns></returns>
public Number Add(Number other)
{
    var targetType = GetOperationTargetType(this, other);
    if (targetType == typeof(decimal))
    {
        return new Number.ToDecimal(null) + other.ToDecimal(null);
    }
    if (targetType == typeof(double))
    {
        return new Number.ToDouble(null) + other.ToDouble(null);
    }
    if (targetType == typeof(float))
    {
        return new Number.ToSingle(null) + other.ToSingle(null);
    }
    if (targetType == typeof(ulong))
    {
        return new Number.ToUInt64(null) + other.ToUInt64(null);
    }
    if (targetType == typeof(long))
    {
        return new Number.ToInt64(null) + other.ToInt64(null);
    }
    if (targetType == typeof(uint))
    {
        return new Number.ToUInt32(null) + other.ToUInt32(null);
    }
    if (targetType == typeof(int))
    {
        return new Number.ToInt32(null) + other.ToInt32(null);
    }
    if (targetType == typeof(ushort))
    {
        return new Number.ToUInt16(null) + other.ToUInt16(null);
    }
    if (targetType == typeof(short))
    {
        return new Number.ToInt16(null) + other.ToInt16(null);
    }
    if (targetType == typeof(byte))
    {
        return new Number.ToByte(null) + other.ToByte(null);
    }
    if (targetType == typeof(sbyte))

```

```
{
    return new Number(ToSByte(null) + other.ToSByte(null));
}
throw new InvalidCastException($"Cannot convert {other.Value}
({other.Value.GetType()}) or {Value} ({Value.GetType()}) to a numeric type");
}
```

The same as over **other** number type the object *Number.cs* is implemented as a readonly struct and is immutable. The code is tested and implemented for:

- netstandard2.0
- netcoreapp2.0 netcoreapp2.1 netcoreapp2.2 netcoreapp3.0
- net46 net461 net462
- net47 net471 net472

A Word on Performance

I recently did some performance tests as I was interested in how much time was spent on casting and evaluation of the right operation type and I was not surprised that the *Number.cs* class is around ~2000 times slower than the native operation. But this was mostly due to unoptimised code. I tested 10.000.000 add operations with the C# native **+** operator on 2 **ints** and done the same with two *Number.cs* classes. I (unsurprisingly) found that 2 major performance problems where the enumeration of the list of **Number.CsFrameworkIntegralTypes**, **Number.CsFrameworkFloatingPointNumberTypes** and to some extent surprisingly for me, the performance of the **Number.Value** getter. So I hard-coded the **Number.GetOperationTargetType** and removed the access to the **Number.Value** and access the value directly via its member variable. That helped a lot and *Number.cs* is now "only" ~140 times slower than a native operation.

Points of Interest

I looked into a lot of code on github and everywhere else and was not able to find something like this general purpose so I decided to write it myself. I will keep developing the *number.cs* class as I might want to support more functions like added support for **Math** functions in the future, so checkout the original file.

History

- 28th June, 2020: Init commit
- 16th Sep, 2020: Updated version to 2.0 containing performance fixes and additional math methods

License


This article, along with any associated source code and files, is licensed under [Microsoft Reciprocal License](#)

About the Author



GerVenson

Software Developer Freelancer

Germany 

I am a Young German Developer.

I was working since 2012 as a Junior Software Developer in the area of WPF with DevExpress and WinForms. I also had some experience with TSQL and Asp.net with AngularJS.

From January 2015 i will be working as an Software Consultant.

From June 2015 i will work as an Freelancer

Comments and Discussions

 **10 messages** have been posted for this article Visit <https://www.codeproject.com/Tips/5272304/Number-cs-for-General-Purpose-Number-Handling-in-C> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#)
[Advertise](#)
[Privacy](#)
[Cookies](#)
[Terms of Use](#)

Article Copyright 2020 by GerVenson
Everything else Copyright © [CodeProject](#),
1999-2020

Web02 2.8.2009011.1