NumberParser

DOI 10.5281/zenodo.887593

Last release -- Master source code

Main page (versión en español) -- Code analysis -- NuGet package -- Video

Introduction

NumberParser (inside the *FlexibleParser* namespace) provides a common framework to deal with all the .NET numeric types. It relies on the following four classes (NumberX):

- *Number* only supports the *decimal* type.
- *NumberD* can support any numeric type via *dynamic*.
- *NumberO* can support different numeric types simultaneously.
- *NumberP* can parse numbers from strings.

```
//1.23m (decimal).
Number number = new Number(1.23m);

//123 (int).
NumberD numberD = new NumberD(123);

//1.23 (decimal). Others: 1 (int) and ' ' (char).
NumberO numberO = new NumberO(1.23m, new Type[] { typeof(int), typeof(char) });

//1 (long).
NumberP numberP = new NumberP("1.23", new ParseConfig(typeof(long)));
```

Common Features

All the NumberX classes have various characteristics in common.

- Defined according to the fields *Value* (*decimal* or *dynamic*) and *BaseTenExponent* (*int*). All of them support ranges beyond [-1, 1] * 10^2147483647.
- Most common arithmetic and comparison operator support.
- Errors managed internally and no exceptions thrown.
- Numerous instantiating alternatives. Implicitly convertible between each other and to related types.

```
//12.3*10^456 (decimal).
Number number = new Number(12.3m, 456);

//123 (int).
Number numberD =
(
   new NumberD(123) < (NumberD)new Number(456) ?
   //123 (int)
   new NumberD(123.456, typeof(int)) :
```

```
   //123.456 (double)
   new NumberD(123.456)
);

//Error (ErrorTypesNumber.InvalidOperation) provoked when dividing by zero.
NumberO numberO = new NumberO(123m, OtherTypes.IntegerTypes) / 0m;

//1234*10^5678 (decimal).
NumberP numberP = (NumberP)"1234e5678";
```

Math2 Class

This class includes all the NumberParser mathematical functionalities.

Custom Functionalities

- *PowDecimal*/*SqrtDecimal* whose *decimal*-based algorithms are more precise than the *System.Math* versions. The whole varocarbas.com Project 10 explains their underlying calculation approach.
- *RoundExact*/*TruncateExact* can deal with multiple rounding/truncating scenarios not supported by the native methods.
- *GetPolynomialFit*/*ApplyPolynomialFit* allow to deal with second degree polynomial fits.
- *Factorial* calculates the factorial of any integer number up to 100000.

```
//158250272872244.91791560253776 (decimal).
Number number = Math2.PowDecimal(123.45m, 6.789101112131415161718m);

//123000 (decimal).
Number number = Math2.RoundExact
(
   123456.789m, 3, RoundType.AlwaysToZero,
   RoundSeparator.BeforeDecimalSeparator
);

//30 (decimal).
NumberD numberD = Math2.ApplyPolynomialFit
(
   Math2.GetPolynomialFit
   (
      new NumberD[] { 1m, 2m, 4m }, new NumberD[] { 10m, 20m, 40m }
   )
   , 3
);

//3628800 (int).
NumberD numberD = Math2.Factorial(10);
```

Native Methods

Math2 also includes *NumberD*-adapted versions of all the *System.Math* methods.

```
//158250289837968.16 (double).
NumberD numberD = Math2.Pow(123.45, 6.789101112131415161718);
```

```
//4.8158362157911885 (double).
NumberD numberD = Math2.Log(123.45m);
```

Further Code Samples

The test application includes a relevant number of descriptive code samples.

Authorship & Copyright

I, Alvaro Carballo Garcia (varocarbas), am the sole author of each single bit of this code.

Equivalently to what happens with all my other online contributions, this code can be considered public domain. For more information about my copyright/authorship attribution ideas, visit the corresponding pages of my sites:

- https://customsolvers.com/copyright/
  ES: https://customsolvers.com/copyright_es/
- http://varocarbas.com/copyright/
  ES: http://varocarbas.com/copyright_es/