Generic math

Article • 04/21/2023

.NET 7 introduces new math-related generic interfaces to the base class library. The availability of these interfaces means you can constrain a type parameter of a generic type or method to be "number-like". In addition, C# 11 and later lets you define static virtual interface members. Because operators must be declared as `static`, this new C# feature lets operators be declared in the new interfaces for number-like types.

Together, these innovations allow you to perform mathematical operations generically—that is, without having to know the exact type you're working with. For example, if you wanted to write a method that adds two numbers, previously you had to add an overload of the method for each type (for example, `static int Add(int first, int second)` and `static float Add(float first, float second)`). Now you can write a single, generic method, where the type parameter is constrained to be a number-like type. For example:

C#

```
static T Add<T>(T left, T right)
    where T : INumber<T>
{
    return left + right;
}
```

In this method, the type parameter `T` is constrained to be a type that implements the new INumber<TSelf> interface. INumber<TSelf> implements the IAdditionOperators<TSelf,TOther,TResult> interface, which contains the + operator. That allows the method to generically add the two numbers. The method can be used with any of .NET's built-in numeric types, because they've all been updated to implement INumber<TSelf> in .NET 7.

Library authors will benefit most from the generic math interfaces, because they can simplify their code base by removing "redundant" overloads. Other developers will benefit indirectly, because the APIs they consume may start supporting more types.

The interfaces

The interfaces were designed to be both fine-grained enough that users can define their own interfaces on top, while also being granular enough that they're easy to consume. To that extent, there are a few core numeric interfaces that most users will interact with,

such as INumber<TSelf> and IBinaryInteger<TSelf>. The more fine-grained interfaces, such as IAdditionOperators<TSelf,TOther,TResult> and ITrigonometricFunctions<TSelf>, support these types and are available for developers who define their own domain-specific numeric interfaces.

- Numeric interfaces
- Operator interfaces
- Function interfaces
- Parsing and formatting interfaces

Numeric interfaces

This section describes the interfaces in System.Numerics that describe number-like types and the functionality available to them.

Interface name	Description
IBinaryFloatingPointIeee754<TSelf>	Exposes APIs common to *binary* floating-point types[1] that implement the IEEE 754 standard.
IBinaryInteger<TSelf>	Exposes APIs common to binary integers[2].
IBinaryNumber<TSelf>	Exposes APIs common to binary numbers.
IFloatingPoint<TSelf>	Exposes APIs common to floating-point types.
IFloatingPointIeee754<TSelf>	Exposes APIs common to floating-point types that implement the IEEE 754 standard.
INumber<TSelf>	Exposes APIs common to comparable number types (effectively the "real" number domain).
INumberBase<TSelf>	Exposes APIs common to all number types (effectively the "complex" number domain).
ISignedNumber<TSelf>	Exposes APIs common to all signed number types (such as the concept of `NegativeOne`).
IUnsignedNumber<TSelf>	Exposes APIs common to all unsigned number types.
IAdditiveIdentity<TSelf,TResult>	Exposes the concept of `(x + T.AdditiveIdentity) == x`.
IMinMaxValue<TSelf>	Exposes the concept of `T.MinValue` and `T.MaxValue`.
IMultiplicativeIdentity<TSelf,TResult>	Exposes the concept of `(x * T.MultiplicativeIdentity) == x`.

[1]The binary floating-point types are Double (`double`), Half, and Single (`float`).

[2]The binary integer types are Byte (`byte`), Int16 (`short`), Int32 (`int`), Int64 (`long`), Int128, IntPtr (`nint`), SByte (`sbyte`), UInt16 (`ushort`), UInt32 (`uint`), UInt64 (`ulong`), UInt128, and UIntPtr (`nuint`).

The interface you're most likely to use directly is INumber<TSelf>, which roughly corresponds to a *real* number. If a type implements this interface, it means that a value has a sign (this includes `unsigned` types, which are considered positive) and can be compared to other values of the same type. INumberBase<TSelf> confers more advanced concepts, such as *complex* and *imaginary* numbers, for example, the square root of a negative number. Other interfaces, such as IFloatingPointIeee754<TSelf>, were created because not all operations make sense for all number types—for example, calculating the floor of a number only makes sense for floating-point types. In the .NET base class library, the floating-point type Double implements IFloatingPointIeee754<TSelf> but Int32 doesn't.

Several of the interfaces are also implemented by various other types, including Char, DateOnly, DateTime, DateTimeOffset, Decimal, Guid, TimeOnly, and TimeSpan.

The following table shows some of the core APIs exposed by each interface.

Interface	API name	Description
IBinaryInteger<TSelf>	`DivRem`	Computes the quotient and remainder simultaneously.
	`LeadingZeroCount`	Counts the number of leading zero bits in the binary representation.
	`PopCount`	Counts the number of set bits in the binary representation.
	`RotateLeft`	Rotates bits left, sometimes also called a circular left shift.
	`RotateRight`	Rotates bits right, sometimes also called a circular right shift.
	`TrailingZeroCount`	Counts the number of trailing zero bits in the binary representation.
IFloatingPoint<TSelf>	`Ceiling`	Rounds the value towards positive infinity. +4.5 becomes +5, and -4.5 becomes -4.
	`Floor`	Rounds the value towards negative infinity. +4.5 becomes +4, and -4.5

Interface	API name	Description
		becomes -5.
	`Round`	Rounds the value using the specified rounding mode.
	`Truncate`	Rounds the value towards zero. +4.5 becomes +4, and -4.5 becomes -4.
IFloatingPointIeee754<TSelf>	`E`	Gets a value representing Euler's number for the type.
	`Epsilon`	Gets the smallest representable value that's greater than zero for the type.
	`NaN`	Gets a value representing `NaN` for the type.
	`NegativeInfinity`	Gets a value representing `-Infinity` for the type.
	`NegativeZero`	Gets a value representing `-Zero` for the type.
	`Pi`	Gets a value representing `Pi` for the type.
	`PositiveInfinity`	Gets a value representing `+Infinity` for the type.
	`Tau`	Gets a value representing `Tau` (`2 * Pi`) for the type.
	(Other)	(Implements the full set of interfaces listed under Function interfaces.)
INumber<TSelf>	`Clamp`	Restricts a value to no more and no less than the specified min and max value.
	`CopySign`	Sets the sign of a specified value to the same as another specified value.
	`Max`	Returns the greater of two values, returning `NaN` if either input is `NaN`.
	`MaxNumber`	Returns the greater of two values, returning the number if one input is `NaN`.
	`Min`	Returns the lesser of two values, returning `NaN` if either input is `NaN`.

Interface	API name	Description
	`MinNumber`	Returns the lesser of two values, returning the number if one input is `NaN`.
	`Sign`	Returns -1 for negative values, 0 for zero, and +1 for positive values.
INumberBase<TSelf>	`One`	Gets the value 1 for the type.
	`Radix`	Gets the radix, or base, for the type. Int32 returns 2. Decimal returns 10.
	`Zero`	Gets the value 0 for the type.
	`CreateChecked`	Creates a value, throwing an OverflowException if the input can't fit.[1]
	`CreateSaturating`	Creates a value, clamping to `T.MinValue` or `T.MaxValue` if the input can't fit.[1]
	`CreateTruncating`	Creates a value from another value, wrapping around if the input can't fit.[1]
	`IsComplexNumber`	Returns true if the value has a non-zero real part and a non-zero imaginary part.
	`IsEvenInteger`	Returns true if the value is an even integer. 2.0 returns `true`, and 2.2 returns `false`.
	`IsFinite`	Returns true if the value is not infinite and not `NaN`.
	`IsImaginaryNumber`	Returns true if the value has a zero real part. This means `0` is imaginary and `1 + 1i` isn't.
	`IsInfinity`	Returns true if the value represents infinity.
	`IsInteger`	Returns true if the value is an integer. 2.0 and 3.0 return `true`, and 2.2 and 3.1 return `false`.
	`IsNaN`	Returns true if the value represents `NaN`.
	`IsNegative`	Returns true if the value is negative. This includes -0.0.
	`IsPositive`	Returns true if the value is positive. This

Interface	API name	Description
		includes 0 and +0.0.
	`IsRealNumber`	Returns true if the value has a zero imaginary part. This means 0 is real as are all `INumber<T>` types.
	`IsZero`	Returns true if the value represents zero. This includes 0, +0.0, and -0.0.
	`MaxMagnitude`	Returns the value with a greater absolute value, returning `NaN` if either input is `NaN`.
	`MaxMagnitudeNumber`	Returns the value with a greater absolute value, returning the number if one input is `NaN`.
	`MinMagnitude`	Returns the value with a lesser absolute value, returning `NaN` if either input is `NaN`.
	`MinMagnitudeNumber`	Returns the value with a lesser absolute value, returning the number if one input is `NaN`.
ISignedNumber<TSelf>	`NegativeOne`	Gets the value -1 for the type.

[1]To help understand the behavior of the three `Create*` methods, consider the following examples.

Example when given a value that's too large:

- `byte.CreateChecked(384)` will throw an OverflowException.
- `byte.CreateSaturating(384)` returns 255 because 384 is greater than Byte.MaxValue (which is 255).
- `byte.CreateTruncating(384)` returns 128 because it takes the lowest 8 bits (384 has a hex representation of `0x0180`, and the lowest 8 bits is `0x80`, which is 128).

Example when given a value that's too small:

- `byte.CreateChecked(-384)` will throw an OverflowException.
- `byte.CreateSaturating(-384)` returns 0 because -384 is smaller than Byte.MinValue (which is 0).
- `byte.CreateTruncating(-384)` returns 128 because it takes the lowest 8 bits (384 has a hex representation of `0xFE80`, and the lowest 8 bits is `0x80`, which is 128).

The `Create*` methods also have some special considerations for IEEE 754 floating-point types, like `float` and `double`, as they have the special values `PositiveInfinity`, `NegativeInfinity`, and `NaN`. All three `Create*` APIs behave as `CreateSaturating`. Also, while `MinValue` and `MaxValue` represent the largest negative/positive "normal" number, the actual minimum and maximum values are `NegativeInfinity` and `PositiveInfinity`, so they clamp to these values instead.

Operator interfaces

The operator interfaces correspond to the various operators available to the C# language.

- They explicitly don't pair operations such as multiplication and division since that isn't correct for all types. For example, `Matrix4x4 * Matrix4x4` is valid, but `Matrix4x4 / Matrix4x4` isn't valid.
- They typically allow the input and result types to differ to support scenarios such as dividing two integers to obtain a `double`, for example, `3 / 2 = 1.5`, or calculating the average of a set of integers.

Interface name	Defined operators
IAdditionOperators<TSelf,TOther,TResult>	`x + y`
IBitwiseOperators<TSelf,TOther,TResult>	`x & y`, `x | y`, `x ^ y`, and `~x`
IComparisonOperators<TSelf,TOther,TResult>	`x < y`, `x > y`, `x <= y`, and `x >= y`
IDecrementOperators<TSelf>	`--x` and `x--`
IDivisionOperators<TSelf,TOther,TResult>	`x / y`
IEqualityOperators<TSelf,TOther,TResult>	`x == y` and `x != y`
IIncrementOperators<TSelf>	`++x` and `x++`
IModulusOperators<TSelf,TOther,TResult>	`x % y`
IMultiplyOperators<TSelf,TOther,TResult>	`x * y`
IShiftOperators<TSelf,TOther,TResult>	`x << y` and `x >> y`
ISubtractionOperators<TSelf,TOther,TResult>	`x - y`
IUnaryNegationOperators<TSelf,TResult>	`-x`
IUnaryPlusOperators<TSelf,TResult>	`+x`

> **Note**
>
> Some of the interfaces define a checked operator in addition to a regular unchecked operator. Checked operators are called in checked contexts and allow a user-defined type to define overflow behavior. If you implement a checked operator, for example, **CheckedSubtraction(TSelf, TOther)**, you must also implement the unchecked operator, for example, **Subtraction(TSelf, TOther)**.

Function interfaces

The function interfaces define common mathematical APIs that apply more broadly than to a specific numeric interface. These interfaces are all implemented by IFloatingPointIeee754<TSelf>, and may get implemented by other relevant types in the future.

Interface name	Description
IExponentialFunctions<TSelf>	Exposes exponential functions supporting `e^x`, `e^x - 1`, `2^x`, `2^x - 1`, `10^x`, and `10^x - 1`.
IHyperbolicFunctions<TSelf>	Exposes hyperbolic functions supporting `acosh(x)`, `asinh(x)`, `atanh(x)`, `cosh(x)`, `sinh(x)`, and `tanh(x)`.
ILogarithmicFunctions<TSelf>	Exposes logarithmic functions supporting `ln(x)`, `ln(x + 1)`, `log2(x)`, `log2(x + 1)`, `log10(x)`, and `log10(x + 1)`.
IPowerFunctions<TSelf>	Exposes power functions supporting `x^y`.
IRootFunctions<TSelf>	Exposes root functions supporting `cbrt(x)` and `sqrt(x)`.
ITrigonometricFunctions<TSelf>	Exposes trigonometric functions supporting `acos(x)`, `asin(x)`, `atan(x)`, `cos(x)`, `sin(x)`, and `tan(x)`.

Parsing and formatting interfaces

Parsing and formatting are core concepts in programming. They're commonly used when converting user input to a given type or displaying a type to the user. These interfaces are in the System namespace.

Interface name	Description
IParsable<TSelf>	Exposes support for `T.Parse(string, IFormatProvider)` and `T.TryParse(string, IFormatProvider, out TSelf)`.

Interface name	Description
ISpanParsable<TSelf>	Exposes support for `T.Parse(ReadOnlySpan<char>, IFormatProvider)` and `T.TryParse(ReadOnlySpan<char>, IFormatProvider, out TSelf)`.
IFormattable[1]	Exposes support for `value.ToString(string, IFormatProvider)`.
ISpanFormattable[1]	Exposes support for `value.TryFormat(Span<char>, out int, ReadOnlySpan<char>, IFormatProvider)`.

[1]This interface isn't new, nor is it generic. However, it's implemented by all number types and represents the inverse operation of `IParsable`.

For example, the following program takes two numbers as input, reading them from the console using a generic method where the type parameter is constrained to be IParsable<TSelf>. It calculates the average using a generic method where the type parameters for the input and result values are constrained to be INumber<TSelf>, and then displays the result to the console.

C#

```
using System.Globalization;
using System.Numerics;

static TResult Average<T, TResult>(T first, T second)
    where T : INumber<T>
    where TResult : INumber<TResult>
{
    return TResult.CreateChecked( (first + second) /
T.CreateChecked(2) );
}

static T ParseInvariant<T>(string s)
    where T : IParsable<T>
{
    return T.Parse(s, CultureInfo.InvariantCulture);
}

Console.Write("First number: ");
var left = ParseInvariant<float>(Console.ReadLine());

Console.Write("Second number: ");
var right = ParseInvariant<float>(Console.ReadLine());

Console.WriteLine($"Result: {Average<float, float>(left, right)}");

/* This code displays output similar to:

First number: 5.0
Second number: 6
```

```
Result: 5.5
*/
```

See also

- Generic math in .NET 7 (blog post)