.NET 7 Preview 5 – Generic Math

Tanner Gooding [MSFT]

June 9th, 2022

In .NET 6 we previewed a feature known as Generic Math. Since then, we have made continuous improvements to the implementation and responded to various feedback from the community in order to ensure that relevant scenarios are possible and the necessary APIs are available.

If you missed out on the original blog post, Generic Math combines the power of generics and a new feature known as `static virtuals in interfaces` to allow .NET developers to take advantage of static APIs, including operators, from generic code. This means that you get all the power of generics, but now with the ability to constrain the input to number like types, so you no longer need to write or maintain many near identical implementations just to support multiple types. It also means that you get access to all your favorite operators and can use them from generic contexts. That is, you can now have `static T Add<T>(T left, T right) where T : INumber<T> => left + right;` where-as previously it would have been impossible to define.

Much like generics, this feature will see the most benefits by API authors where they can simplify the amount of code required they need to maintain. The .NET Libraries did just this to simplify the `Enumerable.Min` and `Enumerable.Max` APIs exposed as part of LINQ. Other developers will benefit indirectly as the APIs they consume may start supporting more types without the requirement for each and every numeric type to get explicit support. Once an API supports `INumber<T>` then it should work with any type that implements the required interface. All devs will likewise benefit from having a more consistent API surface and having more functionality available by default. For example, all types that implement `IBinaryInteger<T>` will support operations like `+` (Addition), `-` (Subtraction), `<<` (Left Shift), and `LeadingZeroCount`.

Generic Math

Lets take a look at an example piece of code that computes a standard deviation. For those unfamiliar, this is a math function used in statistics that builds on two simpler methods: `Sum` and `Average`. It is basically used to determine how spread apart a set of values are.

The first method we'll look at is `Sum`, which just adds a set of values together. The method takes in an `IEnumerable<T>` where `T` must be a type that implements the `INumber<T>` interface. It returns a `TResult` with a similar constraint (it must be a type that implements `INumber<TResult>`). Because two generic parameters are here, it is allowed to return a different type than it takes as an input. This means, for example, you can do `Sum<int, long>` which would allow summing the values of an `int[]` and returning a 64-bit result to help avoid overflow. `TResult.Zero` efficiently gives the value of `0` as a `TResult` and `TResult.CreateChecked` converts `value` from a `T` into a `TResult` throwing an `OverflowException` if it is too large or too small to fit in the destination format. This means, for example, that `Sum<int, byte>` would throw if one of the input values was negative or greater than `255`.

```
public static TResult Sum<T, TResult>(IEnumerable<T> values)
    where T : INumber<T>
    where TResult : INumber<TResult>
{
    TResult result = TResult.Zero;

    foreach (var value in values)
    {
        result += TResult.CreateChecked(value);
    }

    return result;
}
```

The next method is `Average`, which just adds a set of values together (calls `Sum`) and then divides that by the number of values. It doesn't introduce any additional concepts beyond what were used in `Sum`. It does show use of the division operator.

```
public static TResult Average<T, TResult>(IEnumerable<T> values)
    where T : INumber<T>
    where TResult : INumber<TResult>
{
    TResult sum = Sum<T, TResult>(values);
    return TResult.CreateChecked(sum) / TResult.CreateChecked(values.Count());
}
```

`StandardDeviation` is the last method, as indicated above it basically determines how far apart a set of values are. For example, `{ 0, 50, 100 }` has a high deviation of `49.501`; `{ 0, 5, 10 }` on the other hand has a much lower deviation of just `4.5092`. This method introduces a different constraint of `IFloatingPointIeee754` which indicates the return type must be an `IEEE 754` floating-point type such as `double` (`System.Double`) or `float` (`System.Single`). It introduces a new API `CreateSaturating` which explicitly saturates, or clamps, the value on overflow. That is, for `byte.CreateSaturating<int>(value)` it would convert `-1` to `0` because `-1` is less than the minimum value of `0`. It would likewise convert `256` to `255` because `256` is greater than the maximum value of `255`. Saturation is the default behavior for `IEEE 754` floating-point types as they can represent positive and negative infinity as their respective minimum and maximum values. The only other new API is `Sqrt` which behaves just like `Math.Sqrt` or `MathF.Sqrt` and calculates the `square root` of the floating-point value.

```
public static TResult StandardDeviation<T, TResult>(IEnumerable<T> values)
    where T : INumber<T>
    where TResult : IFloatingPointIeee754<TResult>
{
    TResult standardDeviation = TResult.Zero;

    if (values.Any())
    {
        TResult average = Average<T, TResult>(values);
        TResult sum = Sum<TResult, TResult>(values.Select((value) => {
            var deviation = TResult.CreateSaturating(value) - average;
            return deviation * deviation;
        }));
        standardDeviation = TResult.Sqrt(sum / TResult.CreateSaturating(values.Count() - 1));
    }

    return standardDeviation;
}
```

These methods can then be used with any type that implements the required interfaces and in .NET 7 preview 5 we have 20 types that implement these interfaces out of the box. The following table gives a brief description of those types, the corresponding language keyword for C# and F# when that exists, and the primary generic math interfaces they implement. More details on these interfaces and why they exist are provided later on in the Available APIs section.

.NET Type Name	C# Keyword	F# Keyword	Implemented Generic Math Interfaces
System.Byte	byte	byte	IBinaryInteger, IMinMaxValue, IUnsignedNumber
System.Char	char	char	IBinaryInteger, IMinMaxValue, IUnsignedNumber
System.Decimal	decimal	decimal	IFloatingPoint, IMinMaxValue
System.Double	double	float, double	IBinaryFloatingPointIeee754, IMinMaxValue
System.Half			IBinaryFloatingPointIeee754, IMinMaxValue
System.Int16	short	int16	IBinaryInteger, IMinMaxValue, ISignedNumber
System.Int32	int	int	IBinaryInteger, IMinMaxValue, ISignedNumber
System.Int64	long	int64	IBinaryInteger, IMinMaxValue, ISignedNumber

.NET Type Name	C# Keyword	F# Keyword	Implemented Generic Math Interfaces
System.Int128			IBinaryInteger, IMinMaxValue, ISignedNumber
System.IntPtr	nint	nativeint	IBinaryInteger, IMinMaxValue, ISignedNumber
System.Numerics.BigInteger			IBinaryInteger, IUnsignedNumber
System.Numerics.Complex			INumberBase, ISignedNumber
System.Runtime.InteropServices.NFloat			IBinaryFloatingPointIeee754, IMinMaxValue
System.SByte	sbyte	sbyte	IBinaryInteger, IMinMaxValue, ISignedNumber
System.Single	float	float32, single	IBinaryFloatingPointIeee754, IMinMaxValue
System.UInt16	ushort	uint16	IBinaryInteger, IMinMaxValue, IUnsignedNumber
System.UInt32	uint	uint	IBinaryInteger, IMinMaxValue, IUnsignedNumber
System.UInt64	ulong	uint64	IBinaryInteger, IMinMaxValue, IUnsignedNumber
System.UInt128			IBinaryInteger, IMinMaxValue, IUnsignedNumber
System.UIntPtr	nuint	unativeint	IBinaryInteger, IMinMaxValue, IUnsignedNumber

This means that out of the box users get a broad set of support for Generic Math. As the community adopts these interfaces for their own types, the support will continue to grow.

Types Without Language Support

Readers might note that there are a few types here that don't have an entry in the `C# Keyword` or `F# Keyword` column. While these types exist and are supported fully in the BCL, languages like C# and F# do not provide any additional support for them today and so users may surprised when certain language features do not work with them. Some examples are that the language won't provide support for literals (`Int128 value = 0xF_FFFF_FFFF_FFFF_FFFF` isn't valid), constants (`const Int128 Value = 0;` isn't valid), constant folding (`Int128 value = 5;` is evaluated at runtime, not at compile time), or various other functionality that is limited to types that have corresponding language keywords.

The types without language support are:

- `System.Half` is a 16-bit binary floating-point type that implements the IEEE 754 standard much like `System.Double` and `System.Single`. It was originally introduced in .NET 5
- `System.Numerics.BigInteger` is an arbitrary precision integer type and automatically grows to fit the value represented. It was originally introduced in .NET Framework 4.0
- `System.Numerics.Complex` can represent the expression `a + bi` where `a` and `b` are `System.Double` and `i` is the imaginary unit. It was originally introduced in .NET Framework 4.0
- `System.Runtime.InteropServices.NFloat` is a variable precision binary floating-point type that implements the IEEE 754 standard and much like `System.IntPtr` it is 32-bits on a 32-bit platform (equivalent to `System.Single`) and 64-bits on a 64-bit platform (equivalent to `System.Double`) It was originally introduced in .NET 6 and is primarily meant for interop purposes.
- `System.Int128` is a 128-bit signed integer type. It is new in .NET 7
- `System.UInt128` is a 128-bit unsigned integer type. It is new in .NET 7

Breaking Changes Since .NET 6

The feature that went out in .NET 6 was a preview and as such there have been several changes to the API surface based on community feedback. This includes, but is not limited to:

- Renaming `System.IParseable` to `System.IParsable`
- Moving all other new numeric interfaces to the `System.Numerics` namespace
- Introducing `INumberBase` so that types like `System.Numerics.Complex` can be represented
- Splitting the IEEE 754 specific APIs into their own `IFloatingPointIeee754` interface so types like `System.Decimal` can be represented
- Moving various APIs lower in the type hierarchy such as the `IsNaN` or `MaxNumber` APIs
  - Many of the concepts will return a constant value or be a `no-op` on various type
  - Despite this, it is still important that they're available, since the exact type of a generic is unknown and many of these concepts are important for more general algorithms

.NET API reviews are done in the open and are livestreamed for all to view and participate in. Past API review videos can be found on our YouTube channel.

The design doc for the Generic Math feature is available in the dotnet/designs repo on GitHub.

The corresponding PRs updating the document, general discussions around the feature, and links back to the relevant API reviews are also available.

Support in other languages

F# is getting support for static virtuals in interfaces as well and more details should be expected soon in the fsharp/fslang-design repo on GitHub.

A fairly 1-to-1 translation of the C# `Sum` method using the proposed F# syntax is expected to be:

```
let Sum<'T, 'TResult when 'T :> INumber<'T> and 'TResult :> INumber<'TResult>>(values :
IEnumerable<'T>) =
    let mutable result = 'TResult.Zero
    for value in values do
        result <- result 'TResult.CreateChecked(value)
    result
```

Available APIs

Numbers and math are both fairly complex topics and the depth in which one can go is almost without limit. In programming there is often only a loose mapping to the math one may have learned in school and special rules or considerations may exist since execution happens in a system with limited resources. Languages therefore expose many operations that make sense only in the context of certain kinds of numbers or which exist primarily as a performance optimization due to how hardware actually works. The types they expose often have well-defined limits, an explicit layout of the data they are represented by, differing behaviors around rounding or conversions, and more.

Because of this there remains a need to both support numbers in the abstract sense while also still supporting programming specific constructs such as floating-point vs integer, overflow, unrepresentable results; and so it was important as part of designing this feature that the interfaces exposed be both fine-grained enough that users could define their own interfaces built on top while also being granular enough that they were easy to consume. To that extent, there are a few core numeric interfaces that most users will interact with such as `System.Numerics.INumber` and `System.Numerics.IBinaryInteger`; there are then many more interfaces that support these types and support developers defining their own numeric interfaces for their domain such as `IAdditionOperators` and `ITrigonometricFunctions`.

Which interfaces get used will be dependent on the needs of the declaring API and what functionality it relies on. There are a range of powerful APIs exposed to help users efficiently understand the value they've been and decide the appropriate way to work with it including handling edge cases (such as negatives, NaNs, infinities, or imaginary values), having correct conversions (including throwing, saturating, or truncating on overflow), and being extensible enough to version the interfaces moving forward by utilizing Default Interface Methods.

Numeric Interfaces

The types most users will interact with are the `numeric interfaces`. These define the core interfaces describing number-like types and the functionality available to them.

Interface Name	Summary
System.Numerics.IAdditiveIdentity	Exposes the concept of `(x + T.AdditiveIdentity) == x`
System.Numerics.IMinMaxValue	Exposes the concept of `T.MinValue` and `T.MaxValue` (types like BigInteger have no Min/MaxValue)

Interface Name	Summary
System.Numerics.IMultiplicativeIdentity	Exposes the concept of `(x * T.MultiplicativeIdentity) == x`
System.Numerics.IBinaryFloatingPointIeee754	Exposes APIs common to binary floating-point types that implement the IEEE 754 standard
System.Numerics.IBinaryInteger	Exposes APIs common to binary integers
System.Numerics.IBinaryNumber	Exposes APIs common to binary numbers
System.Numerics.IFloatingPoint	Exposes APIs common to floating-point types
System.Numerics.IFloatingPointIeee754	Exposes APIs common to floating-point types that implement the IEEE 754 standard
System.Numerics.INumber	Exposes APIs common to comparable number types (effectively the “Real” number domain)
System.Numerics.INumberBase	Exposes APIs common to all number types (effectively the “Complex” number domain)
System.Numerics.ISignedNumber	Exposes APIs common to all signed number types (such as the concept of `NegativeOne`)
System.Numerics.IUnsignedNumber	Exposes APIs common to all unsigned number types

While there are a few different types here, most users will likely work directly with `INumber<TSelf>`. This roughly corresponds to what some users may recognize as a “real” number and means the value has a sign and well-defined order, making it `IComparable`. `INumberBase<TSelf>` convers more advanced concepts including “complex” and “imaginary” numbers.

Most of the other interfaces, such as `IBinaryNumber`, `IFloatingPoint`, and `IBinaryInteger`, exist because not all operations make sense for all numbers. That is, there are places where APIs only makes sense for values that are known to be binary-based and other places where APIs only make sense for floating-point types. The `IAdditiveIdentity`, `IMinMaxValue`, and `IMultiplicativeIdentity` interfaces exist to cover core properties of number like types. For `IMinMaxValue` in particular, it exists to allow access to the upper (`MaxValue`) and lower (`MinValue`) bounds of a type. Certain types like `System.Numerics.BigInteger` may not have such bounds and therefore do not implement this interface.

`IFloatingPoint<TSelf>` exists to cover both `IEEE 754` types such as `System.Double`, `System.Half`, and `System.Single` as well as other types such as `System.Decimal`. The number of APIs provided by it is much lesser and it is expected most users who explicitly need a floating-point-like type will use `IFloatingPointIeee754`. There is not currently any interface to describe “fixed-point” types but such a definition could exist in the future if there is enough demand.

These interfaces expose APIs previously only available in `System.Math`, `System.MathF`, and `System.Numerics.BitOperations`. This means that functions like `T.Sqrt(value)` are now available to anything implementing `IFloatingPointIeee754<T>` (or more specifically the `IRootFunctions<T>` interface covered below).

Some of the core APIs exposed by each interface includes, but is not limited to the below.

Interface Name	API Name	Summary
IBinaryInteger	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

Interface Name	API Name	Summary
	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	Ceiling	Rounds the value towards positive infinity. +4.5 becomes +5, -4.5 becomes -4
	Floor	Rounds the value towards negative infinity. +4.5 becomes +4, -4.5 becomes -5
	Round	Rounds the value using the specified rounding mode.
	Truncate	Rounds the value towards zero. +4.5 becomes +4, -4.5 becomes -4
IFloatingPointIeee754	E	Gets a value representing Euler's number for the type
	Epsilon	Gets the smallest representable value that is 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

Interface Name	API Name	Summary
	PositiveInfinity	Gets a value representing +Infinity for the type
	Tau	Gets a value representing +Tau, or `2 * Pi` for the type
	–Other–	–Implements the full set of interfaces defined under Functions below–
INumber	Clamp	Restricts a value to no more and no less than the specified min and max value
	CopySign	Sets the sign of a give 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
	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	One	Gets the value 1 for the type
	Radix	Gets the radix, or base, for the type. Int32 returns 2. Decimal returns 10

Interface Name	API Name	Summary
	Zero	Gets the value `0` for the type
	CreateChecked	Creates a value from another value, throwing if the other value can't be represented
	CreateSaturating	Creates a value from another value, saturating if the other value can't be represented
	CreateTruncating	Creates a value from another value, truncating if the other value can't be represented
	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, 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` is not
	IsInfinity	Returns true if the value represents infinity.
	IsInteger	Returns true if the value is an integer. 2.0 and 3.0 return true, 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

Interface Name	API Name	Summary
	IsPositive	Returns true if the value is positive, this 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	NegativeOne	Gets the value `-1` for the type

Functions

The function interfaces define common mathematical APIs that may be more broadly applicable than to a specific numeric interface. They are currently all implemented by `IFloatingPointIeee754` and may also get implemented by other relevant types in the future.

Interface Name	Summary
System.Numerics.IExponentialFunctions	Exposes exponential functions supporting `e^x`, `e^x - 1`, `2^x`, `2^x - 1`, `10^x`, and `10^x - 1`
System.Numerics.IHyperbolicFunctions	Exposes hyperbolic functions supporting `acosh(x)`, `asinh(x)`, `atanh(x)`, `cosh(x)`, `sinh(x)`, and `tanh(x)`
System.Numerics.ILogarithmicFunctions	Exposes logarithmic functions supporting `ln(x)`, `ln(x + 1)`, `log2(x)`, `log2(x + 1)`, `log10(x)`, and `log10(x + 1)`

Interface Name	Summary
System.Numerics.IPowerFunctions	Exposes power functions supporting `x^y`
System.Numerics.IRootFunctions	Exposes root functions supporting `cbrt(x)` and `sqrt(x)`
System.Numerics.ITrigonometricFunctions	Exposes trigonometric functions supporting `acos(x)`, `asin(x)`, `atan(x)`, `cos(x)`, `sin(x)`, and `tan(x)`

Parsing and Formatting

Parsing and formatting are core concepts in programming. They are typically used to support converting user input to a given type or to display a given type to the user.

Interface Name	Summary
System.IFormattable	Exposes support for `value.ToString(string, IFormatProvider)`
System.ISpanFormattable	Exposes support for `value.TryFormat(Span<char>, out int, ReadOnlySpan<char>, IFormatProvider)`
System.IParsable	Exposes support for `T.Parse(string, IFormatProvider)`
System.ISpanParsable	Exposes support for `T.Parse(ReadOnlySpan<char>, IFormatProvider)`

Operators

Central to Generic Math is the ability to expose operators as part of an interface. .NET 7 provides the following interfaces which expose the core operators supported by most languages. This also includes new functionality in the form of `user-defined checked operators` and `unsigned right shift`.

Interface Name	Summary
System.Numerics.IAdditionOperators	Exposes the `x + y` and `checked(x + y)` operators
System.Numerics.IBitwiseOperators	Exposes the `x & y`, `x | y`, `x ^ y`, and `~x` operators
System.Numerics.IComparisonOperators	Exposes the `x < y`, `X > y`, `x <= y`, and `x >= y` operators
System.Numerics.IDecrementOperators	Exposes the `--x`, `checked(--x)`, `x--`, and `checked(x--)` operators
System.Numerics.IDivisionOperators	Exposes the `x / y` and `checked(x / y)` operators
System.Numerics.IEqualityOperators	Exposes the `x == y` and `x != y` operators
System.Numerics.IIncrementOperators	Exposes the `++x`, `checked(++x)`, `x++`, and `checked(x++)` operators
System.Numerics.IModulusOperators	Exposes the `x % y` operator
System.Numerics.IMultiplyOperators	Exposes the `x * y` and `checked(x * y)` operators
System.Numerics.IShiftOperators	Exposes the `x << y`, `x >> y`, and `x >>> y` operators
System.Numerics.ISubtractionOperators	Exposes the `x - y` and `checked(x - y)` operators
System.Numerics.IUnaryNegationOperators	Exposes the `-x` and `checked(-x)` operators
System.Numerics.IUnaryPlusOperators	Exposes the `+x` operator

User-Defined Checked Operators

`User-defined checked operators` allow a different implementation to be provided which will throw `System.OverflowException` rather than silently truncating their result. These alternative implementations are available to C# code by using the `checked` keyword or setting `<CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>` in your project settings. The versions that truncate are available by using the `unchecked` keyword or ensuring `CheckForOverflowUnderflow` is `false` (this is the default experience for new projects).

Some types, such as floating-point types, may not have differing behavior as they `saturate` to `PositiveInfinity` and `NegativeInfinity` rather than truncating. `BigInteger` is another type that does not have differing behavior between the unchecked and checked versions of the operators as the type simply grows to fit the value. 3rd party types may also have their own unique behavior.

Developers can declare their own `user-defined checked operators` by placing the `checked` keyword after the `operator` keyword. For example, `public static Int128 operator checked +(Int128 left, Int128 right)` declares a `checked addition` operator and `public static explicit operator checked int(Int128 value)` declares a `checked explicit conversion` operator.

Unsigned Right Shift

Unsigned right shift (`>>>`) allows shifting to occur that doesn't carry the sign. That is, for `-8 >> 2` the result is `-2` while `-8 >>> 2` is `+1073741822`.

This is somewhat easier to visualize when looking at the hexadecimal or binary representation. For `x >> y` the sign of the value is preserved and so for positive values `0` is shifted in while for negative values `1` is shifted in instead. However, for `x >>> y` the sign of the value is ignored and `0` is always shifted in. This is similar to first casting the value to an `unsigned` type of the same sign and then doing the shift, that is it is similar to `(int)((uint)x >> y)` for `int`.

Expression	Decimal	Hexadecimal	Binary
`-8`	-8	0xFFFF_FFF8	0b1111_1111_1111_1111_1111_1111_1111_1000
`-8 >> 2`	-2	0xFFFF_FFFE	0b1111_1111_1111_1111_1111_1111_1111_1110
`-8 >>> 2`	+1,073,741,822	0x3FFF_FFFE	0b0011_1111_1111_1111_1111_1111_1111_1110

Closing

The amount of functionality now available in a generic context is quite large, allowing your code to be simpler, more maintainable, and more expressive. Generic Math will empower every developer to achieve more, and we are excited to see how you decide to utilize it!

Tanner Gooding [MSFT] Software Engineer, .NET Team
Follow