# Tutorial: Explore C# 11 feature - static virtual members in interfaces

Article • 08/03/2022

C# 11 and .NET 7 include *static virtual members in interfaces*. This feature enables you to define interfaces that include overloaded operators or other static members. Once you've defined interfaces with static members, you can use those interfaces as constraints to create generic types that use operators or other static methods. Even if you don't create interfaces with overloaded operators, you'll likely benefit from this feature and the generic math classes enabled by the language update.

In this tutorial, you'll learn how to:

- ✓ Define interfaces with static members.
- ✓ Use interfaces to define classes that implement interfaces with operators defined.
- ✓ Create generic algorithms that rely on static interface methods.

## Prerequisites

You'll need to set up your machine to run .NET 7, which supports C# 11. The C# 11 compiler is available starting with Visual Studio 2022, version 17.3 or the .NET 7 SDK.

## Static abstract interface methods

Let's start with an example. The following method returns the midpoint of two `double` numbers:

C#

```
public static double MidPoint(double left, double right) =>
    (left + right) / (2.0);
```

The same logic would work for any numeric type: `int`, `short`, `long`, `float` `decimal`, or any type that represents a number. You need to have a way to use the `+` and `/` operators, and to define a value for `2`. You can use the System.Numerics.INumber<TSelf> interface to write the preceding method as the following generic method:

C#

```
public static T MidPoint<T>(T left, T right)
    where T : INumber<T> => (left + right) / T.CreateChecked(2);  //
note: the addition of left and right may overflow here; it's just for
demonstration purposes
```

Any type that implements the INumber<TSelf> interface must include a definition for `operator +`, and for `operator /`. The denominator is defined by `T.CreateChecked(2)` to create the value `2` for any numeric type, which forces the denominator to be the same type as the two parameters. INumberBase<TSelf>.CreateChecked<TOther>(TOther) creates an instance of the type from the specified value and throws an OverflowException if the value falls outside the representable range. (This implementation has the potential for overflow if `left` and `right` are both large enough values. There are alternative algorithms that can avoid this potential issue.)

You define static abstract members in an interface using familiar syntax: You add the `static` and `abstract` modifiers to any static member that doesn't provide an implementation. The following example defines an `IGetNext<T>` interface that can be applied to any type that overrides `operator ++`:

C#

```
public interface IGetNext<T> where T : IGetNext<T>
{
    static abstract T operator ++(T other);
}
```

The constraint that the type argument, `T`, implements `IGetNext<T>` ensures that the signature for the operator includes the containing type, or its type argument. Many operators enforce that its parameters must match the type, or be the type parameter constrained to implement the containing type. Without this constraint, the `++` operator couldn't be defined in the `IGetNext<T>` interface.

You can create a structure that creates a string of 'A' characters where each increment adds another character to the string using the following code:

C#

```
public struct RepeatSequence : IGetNext<RepeatSequence>
{
    private const char Ch = 'A';
    public string Text = new string(Ch, 1);
```

```
    public RepeatSequence() {}

    public static RepeatSequence operator ++(RepeatSequence other)
        => other with { Text = other.Text + Ch };

    public override string ToString() => Text;
}
```

More generally, you can build any algorithm where you might want to define `++` to mean "produce the next value of this type". Using this interface produces clear code and results:

C#

```
var str = new RepeatSequence();

for (int i = 0; i < 10; i++)
    Console.WriteLine(str++);
```

The preceding example produces the following output:

PowerShell

```
A
AA
AAA
AAAA
AAAAA
AAAAAA
AAAAAAA
AAAAAAAA
AAAAAAAAA
AAAAAAAAAA
```

This small example demonstrates the motivation for this feature. You can use natural syntax for operators, constant values, and other static operations. You can explore these techniques when you create multiple types that rely on static members, including overloaded operators. Define the interfaces that match your types' capabilities and then declare those types' support for the new interface.

# Generic math

The motivating scenario for allowing static methods, including operators, in interfaces is to support generic math algorithms. The .NET 7 base class library contains interface definitions for many arithmetic operators, and derived interfaces that combine many

arithmetic operators in an `INumber<T>` interface. Let's apply those types to build a `Point<T>` record that can use any numeric type for `T`. The point can be moved by some `XOffset` and `YOffset` using the `+` operator.

Start by creating a new Console application, either by using `dotnet new` or Visual Studio. Set the C# language version to "preview", which enables C# 11 preview features. Add the following element to your *csproj* file inside a `<PropertyGroup>` element:

XML

```xml
<LangVersion>preview</LangVersion>
```

> **Note**
>
> This element cannot be set using the Visual Studio UI. You need to edit the project file directly.

The public interface for the `Translation<T>` and `Point<T>` should look like the following code:

C#

```
// Note: Not complete. This won't compile yet.
public record Translation<T>(T XOffset, T YOffset);

public record Point<T>(T X, T Y)
{
    public static Point<T> operator +(Point<T> left, Translation<T>
right);
}
```

You use the `record` type for both the `Translation<T>` and `Point<T>` types: Both store two values, and they represent data storage rather than sophisticated behavior. The implementation of `operator +` would look like the following code:

C#

```
public static Point<T> operator +(Point<T> left, Translation<T>
right) =>
    left with { X = left.X + right.XOffset, Y = left.Y + right.YOff-
set };
```

For the previous code to compile, you'll need to declare that `T` supports the `IAdditionOperators<TSelf, TOther, TResult>` interface. That interface includes the `operator +` static method. It declares three type parameters: One for the left operand, one for the right operand, and one for the result. Some types implement `+` for different operand and result types. Add a declaration that the type argument, `T` implements `IAdditionOperators<T, T, T>`:

C#

```
public record Point<T>(T X, T Y) where T : IAdditionOperators<T, T,
T>
```

After you add that constraint, your `Point<T>` class can use the `+` for its addition operator. Add the same constraint on the `Translation<T>` declaration:

C#

```
public record Translation<T>(T XOffset, T YOffset) where T :
IAdditionOperators<T, T, T>;
```

The `IAdditionOperators<T, T, T>` constraint prevents a developer using your class from creating a `Translation` using a type that doesn't meet the constraint for the addition to a point. You've added the necessary constraints to the type parameter for `Translation<T>` and `Point<T>` so this code works. You can test by adding code like the following above the declarations of `Translation` and `Point` in your *Program.cs* file:

C#

```
var pt = new Point<int>(3, 4);

var translate = new Translation<int>(5, 10);

var final = pt + translate;

Console.WriteLine(pt);
Console.WriteLine(translate);
Console.WriteLine(final);
```

You can make this code more reusable by declaring that these types implement the appropriate arithmetic interfaces. The first change to make is to declare that `Point<T, T>` implements the `IAdditionOperators<Point<T>, Translation<T>, Point<T>>` interface. The `Point` type makes use of different types for operands and the result. The

`Point` type already implements an `operator +` with that signature, so adding the interface to the declaration is all you need:

C#

```
public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>,
Translation<T>, Point<T>>
    where T : IAdditionOperators<T, T, T>
```

Finally, when you're performing addition, it's useful to have a property that defines the additive identity value for that type. There's a new interface for that feature: IAdditiveIdentity<TSelf,TResult>. A translation of `{0, 0}` is the additive identity: The resulting point is the same as the left operand. The `IAdditiveIdentity<TSelf, TResult>` interface defines one readonly property, `AdditiveIdentity`, that returns the identity value. The `Translation<T>` needs a few changes to implement this interface:

C#

```
using System.Numerics;

public record Translation<T>(T XOffset, T YOffset) :
IAdditiveIdentity<Translation<T>, Translation<T>>
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>
{
    public static Translation<T> AdditiveIdentity =>
        new Translation<T>(XOffset: T.AdditiveIdentity, YOffset:
T.AdditiveIdentity);
}
```

There are a few changes here, so let's walk through them one by one. First, you declare that the `Translation` type implements the `IAdditiveIdentity` interface:

C#

```
public record Translation<T>(T XOffset, T YOffset) :
IAdditiveIdentity<Translation<T>, Translation<T>>
```

You next might try implementing the interface member as shown in the following code:

C#

```
public static Translation<T> AdditiveIdentity =>
    new Translation<T>(XOffset: 0, YOffset: 0);
```

The preceding code won't compile, because `0` depends on the type. The answer: Use `IAdditiveIdentity<T>.AdditiveIdentity` for `0`. That change means that your constraints must now include that `T` implements `IAdditiveIdentity<T>`. That results in the following implementation:

C#

```
public static Translation<T> AdditiveIdentity =>
    new Translation<T>(XOffset: T.AdditiveIdentity, YOffset:
T.AdditiveIdentity);
```

Now that you've added that constraint on `Translation<T>`, you need to add the same constraint to `Point<T>`:

C#

```
using System.Numerics;

public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>,
Translation<T>, Point<T>>
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>
{
    public static Point<T> operator +(Point<T> left, Translation<T>
right) =>
        left with { X = left.X + right.XOffset, Y = left.Y + right.Y-
Offset };
}
```

This sample has given you a look at how the interfaces for generic math compose. You learned how to:

- ✓ Write a method that relied on the `INumber<T>` interface so that method could be used with any numeric type.
- ✓ Build a type that relies on the addition interfaces to implement a type that only supports one mathematical operation. That type declares its support for those same interfaces so it can be composed in other ways. The algorithms are written using the most natural syntax of mathematical operators.

Experiment with these features and register feedback. You can use the *Send Feedback* menu item in Visual Studio, or create a new issue in the roslyn repository on GitHub. Build generic algorithms that work with any numeric type. Build algorithms using these interfaces where the type argument may only implement a subset of number-like capabilities. Even if you don't build new interfaces that use these capabilities, you can experiment with using them in your algorithms.

# See also

- Generic math