Upcoming Features in C#

Mads Torgersen, MSFT

This document describes the C# language features currently featured in the March CTP. There’s also discussion of a few of the VB features in the preview, where those are intended to be eventually implemented in C# as well.

Note that we have more language features planned that are not yet implemented. So look out for more to come in future CTPs and eventually a shipping version of C#.

Contents

[1 Initializers for auto-properties 2](#_Toc382559013)

[2 Primary constructors 2](#_Toc382559014)

[2.1 Parameters on classes and structs 2](#_Toc382559015)

[2.2 Field parameters 3](#_Toc382559016)

[2.3 Explicit constructors 4](#_Toc382559017)

[2.4 Base initializer 4](#_Toc382559018)

[2.5 Partial types 4](#_Toc382559019)

[3 Using static 4](#_Toc382559020)

[3.1 Extension methods 4](#_Toc382559021)

[4 Declaration expressions 5](#_Toc382559022)

[5 Exception filters 6](#_Toc382559023)

[6 Binary literals and digit separators (Try them out in VB) 6](#_Toc382559024)

[7 Indexed members and element initializers 6](#_Toc382559025)

[8 Await in catch and finally blocks 7](#_Toc382559026)

[9 Extension Add methods in collection initializers 7](#_Toc382559027)

# Initializers for auto-properties

You can now add an initializer to an auto-property, just as you can a field:

public class Customer  
{  
 public string First { get; set; } = "Jane";  
 public string Last { get; set; } = "Doe";  
}

The initializer directly initializes the backing field; it doesn’t work through the setter of the auto-property. For this reason it now makes sense to have auto-properties *without* setters:

public class Customer  
{  
 public string First { get; } = "Jane";  
 public string Last { get; } = "Doe";  
}

Getter-only auto-props are allowed as long as there’s an initializer. Otherwise it would never be able to contain any other value than the default value of the type.

Just like field initializers, auto-property initializers cannot reference ‘this’ – after all they are executed before the object is properly initialized. This would mean that there aren’t a whole lot of interesting choices for what to initialize the auto-properties *to*. Especially getter-only auto-properties would seem kind of useless if they couldn’t somehow acquire values passed in to constructors of the object. However, *primary constructors* change that. Auto-property initializers and primary constructors thus enhance each other.

# Primary constructors

Primary constructors allow constructor parameters to be declared directly on the class or struct, without an explicit constructor declaration in the body of the type declaration. These parameters are in scope as simple names for initialization throughout the whole class declaration.

*Note: We changed the design, and the following describes the new design. However, the changes didn’t make it in in time for the preview build, so you’ll still see the old semantics: parameters are* *captured implicitly and there is no syntax for explicit capture.*

## Parameters on classes and structs

Here’s an example of a class with a primary constructor:

public class Customer(string first, string last)  
{  
 public string First { get; } = first;  
 public string Last { get; } = last;  
}

Achieving the same effect without primary constructors would have required private fields to hold the values of first and last, an explicit constructor to initialize them, and a getter body in First and Last to expose them:

public class Customer  
{  
 private string first;  
 private string last;

public Customer(string first, string last)  
 {  
 this.first = first;  
 this.last = last;  
 }

public string First { get { return first; } }  
 public string Last { get { return last; } }  
}

Just to give you a sense of how much boilerplate this saves, I’ve highlighted the unnecessary chunks that are removed by the use of primary constructor and initialized getter-only auto-properties.

This is about expressing types more concisely, but note that it also removes an important difference in the language between mutable and immutable types: auto-properties were a shorthand available only if you were willing to make your class mutable, and so the temptation to default to that was great. Now, with getter-only auto-properties, the playing field has been leveled between mutable and immutable.

There is currently no way to specify an explicit constructor body for a primary constructor. *We didn’t land on a good syntax for this, and are not sure how important it is, since most of constructor bodies are usually initialization anyway. However, many have brought up constructor argument validation as something you’d often want to express. What do you think?*

## Field parameters

*Note: Field parameters don’t work in the current prototype.*

By default, a primary constructor parameter is only around at initialization time. Function members such as properties and methods cannot refer to it, because by the time they are called, the object has been constructed and the parameter is gone.

We considered (and experimented with) letting parameters be implicitly “captured” into compiler generated private fields if they were used after construction, but that lead to wasteful and buggy code where parameters were too often unintentionally captured into fields.

Of course you can declare your own private field and initialize it with a parameter:

public class Customer(string first, string last, DateTime bday)  
{  
 public string First { get; } = first;  
 public string Last { get; } = last;

private DateTime \_bday = bday;  
}

Unfortunately the private field wouldn’t be able to use the same name as the parameter, and needs an alternative name, e.g. with an underbar in front.

To avoid this, we give you syntax to *explicitly* capture primary constructor parameters into fields:

public class Customer(string first, string last, private DateTime bday)  
{  
 public string First { get; } = first;  
 public string Last { get; } = last;  
}

When you put an accessibility modifier (most likely private) on a primary constructor parameter it means make a parameter *and* a field with the same name, and initialize the field with the parameter. This way, you get to reference it in methods and properties, not just during initialization:

public int Age { get { return ComputeAge(bday); } }

## Explicit constructors

A class declaration with a primary constructor can still define other constructors. However, to ensure that arguments actually get passed to the primary constructor parameters, all other constructors must call a this(…) initializer:

public Point() : this(0, 0) {} // calls the primary constructor

Explicit constructors can call each other, but will directly or indirectly end up calling the primary constructor, as it is the only one that can call a base(…) initializer.

In structs with primary constructors, explicit constructors furthermore cannot chain to the parameterless default constructor: they must directly or indirectly chain to the primary constructor.

## Base initializer

The primary constructor always implicitly or explicitly calls a base initializer. If no base initializer is specified, just like all constructors it will default to calling a parameterless base constructor.

The way to explicitly call the base initializer is to pass an argument list to a base class specifier:

class BufferFullException() : Exception("Buffer full") { … }

## Partial types

If a type is declared in multiple parts, only one part can declare primary constructor parameters, and only that part can declare arguments in its base class specification.

If specified, primary constructor parameters are in scope in all parts of the type.

# Using static

*Note:* The current implementation of this feature does not reflect the current design. More details below.

The feature allows specifying a type in a using clause, making all the accessible static members of the type available without qualification in subsequent code:

using System.Console;  
using System.Math;

class Program  
{  
 static void Main()  
 {  
 WriteLine(Sqrt(3\*3 + 4\*4));   
 }  
}

This feature clearly has the ability to make your code shorter. There’s a concern that it can clutter up your namespace, and cause weird ambiguities, especially with classes that weren’t authored to be “opened up” like that. How many Create(…) methods can one handle without being confused about what you’re creating?

## Extension methods

Extension methods are static methods. This raises a question about how a using static should bring them in. As a top level method in scope? As an extension method? Both? Let’s look at an example:

using System.Linq.Enumerable; // Just the type, not the whole namespace

class Program  
{  
 static void Main()  
 {  
 var range = Range(5, 17); // (1)

var odd = Where(range, i => i % 2 == 1); // (2)

var even = range.Where(i => i % 2 == 0); // (3)  
 }  
}

Range is an ordinary static method so the call marked (1) is and should be allowed. The current implementation doesn’t do anything special for extension methods, so it allows the call to Where as a static method in (2), but not as an extension method in (3).

This is not what we intend. Allowing (3) would give us a long asked-for way to bring in extension methods more selectively per type they are defined in, rather than just per namespace. On the other hand, extension methods were rarely designed to be used as static methods (that is just a fallback for the rare case where there are ambiguities), and we do not really want them to clutter up the top-level namespace. So we want to disallow (2). We hope to make these changes in a future CTP.

# Declaration expressions

Declaration expressions allow you to declare local variables in the middle of an expression, with or without an initializer. Here are some examples:

if (int.TryParse(s, out int i)) { … }

GetCoordinates(out var x, out var y);

Console.WriteLine("Result: {0}", (int x = GetValue()) \* x);

if ((string s = o as string) != null) { … s … }

This is particularly useful for out parameters, where you no longer have to declare variables on a separate line to pass in. This is neat even in the best of cases, but in some scenarios where only expressions are allowed, it is necessary for using out parameters. In query clauses for example:

from s in strings  
select int.TryParse(s, out int i) ? i : -1;

Intuitively, the scope of variables declared in an expression extends out to the nearest block, structured statement (such as if or while) or embedded statement (such as a branch or body of an if or while). Also, lambda bodies, query clauses and field/property initializers act as scope boundaries.

In most positions, declaration expressions allow var only if there’s an initializer to infer the type from, just like declaration statements. However, if a declaration expression is passed as an out parameter, we will try to do overload resolution without the type of that argument, and then infer the type from the selected method:

void GetCoordinates(out int x, out int y) { … }

GetCoordinates(out var x, out var y); // we’ll infer int

Declaration expressions take a little getting used to, and can probably be abused in new and interesting ways. Take them for a spin and let us know what you think.

# Exception filters

VB has them. F# has them. Now C# has them too. This is what they look like:

try { … }

catch (MyException e) if (myfilter(e))  
{  
 …  
}

If the parenthesized expression evaluates to true, the catch block is run, otherwise the exception keeps going.

Exception filters are preferable to catching and rethrowing because they leave the stack unharmed. If the exception later causes the stack to be dumped, you can see where it originally came from, rather than just the last place it was rethrown.

# Binary literals and digit separators (Try them out in VB)

We want to allow binary literals in C#. No longer will you need to belong to the secret brotherhood of Hex in order to specify bit vectors or flags values!

var bits = 0b00101110;

For long literals (and these new binary ones can easily get long!) being able to specify digits in groups also seems useful. C# will allow an underbar ‘\_’ in literals to separate such groups:

var bits = 0b0010\_1110;

var hex = 0x00\_2E;

var dec = 1\_234\_567\_890;

You can put as many as you want, wherever you want, except at the front.

# Indexed members and element initializers

Object and collection initializers are useful for declaratively initializing fields and properties of objects, or giving a collection an initial set of elements. Initializing dictionaries is less elegant. We are adding a new syntax to object initializers allowing you to set values to keys through any indexer that the new object has:

var numbers = new Dictionary<int, string> {  
 [7] = "seven",  
 [9] = "nine",  
 [13] = "thirteen"  
};

Specifically if the keys are strings, you sometimes want to think of them almost as a weak kind of members of the object. This is the case for instance if you have semi-structured data in wire formats such as JSON, that is “object-like” but not strongly typed in C# itself. We are adding a syntax for “indexed members”, which let you pretend that a literal string key is a sort of member:

var customer = new JsonData {  
 $first = "Anders", // => ["first"] = "Anders"  
 $last = "Hejlsberg" // => ["last"] = "Hejlsberg"  
}

string first = customer.$first; // => customer["first"]

Indexed members are indicated by a $ sign, and can be used in object initializers and member access. They just translate down to indexing with string literals, so they can be used wherever the receiving object has an indexer that takes a single string index.

# Await in catch and finally blocks

In C# 5.0 we don’t allow the await keyword in catch and finally blocks, because we’d somehow convinced ourselves that it wasn’t possible to implement. Now we’ve figured it out, so apparently it wasn’t impossible after all.

This has actually been a significant limitation, and people have had to employ unsightly workarounds to compensate. That is no longer necessary:

Resource res = null;

try  
{  
 res = await Resource.OpenAsync(…); // You could do this.  
 …  
}   
catch(ResourceException e)  
{  
 await Resource.LogAsync(res, e); // Now you can do this …  
}  
finally  
{  
 if (res != null) await res.CloseAsync(); // … and this.  
}

The implementation *is* quite complicated, but you don’t have to worry about that. That’s the whole point of having async in the language.

# Extension Add methods in collection initializers

When we first implemented collection initializers in C#, the Add methods that get called couldn’t be extension methods. VB got it right from the start, but it seems we forgot about it in C#. This has been fixed: the code generated from a collection initializer will now happily call an extension method called Add. It’s not much of a feature, but it’s occasionally useful, and it turned out implementing it in the new compiler amounted to removing a check that prevented it.