Upcoming Features in C#

Mads Torgersen, MSFT

This document describes language features currently planned for the next version of C#. Some of these are already implemented and available in the current CTP, others are yet to come. It is also likely that some of these features will not make it in due to resource constraints or other change of plans.

Note that some of these features currently only work when the LangVersion of the compiler is set to "experimental", e.g. by adding the following line to the csproj file in appropriate places (e.g. next to <WarnLevel> directives):

```
        <LangVersion>experimental</LangVersion>
```

Contents

1 Auto-property enhancements

1.1 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. The initializers are executed in order as written, just as – and along with – field initializers.

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*. However, *primary constructors* change that. Auto-property initializers and primary constructors thus enhance each other.

1.2 Getter-only auto-properties

Auto-properties can now be declared without a setter.

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

The backing field of a getter-only auto-property is implicitly declared as `readonly` (though this matters only for reflection purposes). It can be initialized through an initializer on the property as in the example above. Also, a getter-only property can be assigned to in the declaring type's constructor body, which causes the value to be assigned directly to the underlying field:

```
public class Customer
{
    public string Name { get; };

    public Customer(string first, string last)
    {
        Name = first + " " + last;
    }
}
```

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.

Note: In the current CTP getter-only properties can only be initialized with initializers, not from within a constructor body.

2 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.

2.1 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 and getter-only auto-properties with initializers 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.

2.2 Primary constructor bodies

Most of what goes on in constructor bodies today is initialization of fields and auto-properties. With primary constructors and auto-property initializers, there is no longer a big need for this. However, fairly often there is a desire to do other things in constructor bodies, such as validate the constructor arguments.

For these cases it is possible to specify a primary constructor body simply by enclosing it in curly braces inside of the declared type:

```
public class Customer(string first, string last)
{
    {
        if (first == null) throw new ArgumentNullException("first");
        if (last == null) throw new ArgumentNullException("last");
    }

    public string First { get; } = first;
    public string Last { get; } = last;
}
```

2.3 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.

2.4 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") { … }
```

2.5 Initialization scope

Primary constructor parameters live in a scope of their own, which spans those base arguments as well as initializer expressions on the members, and is nested inside of the type body itself. It is perfectly fine to have primary constructors with the same name as members of the type:

```
public class Customer(string first, string last, long id) :
    Person(first, last)
{
    private long id = id;
}
```

2.6 Partial types

If a type is declared in multiple parts, only one part can declare primary constructor *parameter*s, and only that part can declare *argument*s in its base class specification.

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

3 Expression bodied function members

Lambda expressions can be declared with an expression body as well as a conventional function body consisting of a block. This feature brings the same convenience to function members of types.

3.1 Expression bodies on method-like members

Methods as well as user-defined operators and conversions can be given an expression body by use of the “lambda arrow”:

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public static implicit operator string(Person p)
    => p.First + " " + p.Last;
```

The effect is exactly the same as if the methods had had a block body with a single return statement.

For void returning methods – and `Task` returning async methods – the arrow syntax still applies, but the expression following the arrow must be a statement expression (just as is the rule for lambdas):

```
public void Print() => Console.WriteLine(First + " " + Last);
```

3.2 Expression bodies on property-like function members

Properties and indexers can have getters and setters. Expression bodies can be used to write getter-only properties and indexers where the body of the getter is given by the expression body:

```
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

Note that there is no `get` keyword: It is implied by the use of the expression body syntax.

Note: In the current CTP, expression bodied indexers don't work properly yet.

4 Initializers in structs

To help facilitate primary constructor usage, structs are now allowed to have initializers. There's a niggle that the default (parameterless) constructor does not run the initializers. That's a consequence of the default constructor being special and not being able to run user-defined code. Only user-defined constructors run the initializers.

As a consequence, initializers are only allowed when the struct has at least one user defined constructor. Otherwise there'd be no way for the initializers to run.

```
public struct Tally(int count, int sum)
{
    public int Count { get; } = count;
    public int Sum { get; } = sum;
    public double Average => (double)Sum / Count;
}
```

5 Using static

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

```
using System.Console;
using System.Math;

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

This is great for when you have a set of functions related to a certain domain that you use all the time. `System.Math` would be a common example of that.

5.1 Extension methods

Note: The current implementation does not implement the functionality for extension methods as described here. Instead it treats extension methods just like ordinary static methods.

Extension methods are static methods, but are intended to be used as instance methods. Instead of bringing extension methods into the global scope, the using static feature makes the extension methods of the type available as extension methods:

```
using System.Linq.Enumerable; // Just the type, not the whole namespace
class Program
{
    static void Main()
    {
        var range = Range(5, 17);                // Ok: not extension
        var odd = Where(range, i => i % 2 == 1); // Error, not in scope
        var even = range.Where(i => i % 2 == 0); // Ok
    }
}
```

This does mean that it can now be a breaking change to turn an ordinary static method into an extension method, which was not the case before. But extension methods are generally only called as static methods in the rare cases where there is an ambiguity. In those cases, it seems right to require full qualification of the method anyway.

6 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.

It is also a common and accepted form of “abuse” to use exception filters for side effects; e.g. logging. They can inspect an exception “flying by” without intercepting its course. In those cases, the filter will often be a call to a false-returning helper function which executes the side effects:

```
private static bool Log(Exception e) { /* log it */ ; return false; }
…
try { … } catch (Exception e) if (Log(e)) {}
```

7 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
```

8 Nameof expressions

Occasionally you need to provide a string that names some program element: when throwing an `ArgumentException` you want to name the guilty argument; when raising a `PropertyChanged` event you want to name the property that changed, etc.

Using string literals for this purpose is simple, but error prone. You may spell it wrong, or a refactoring may leave it stale. Nameof expressions are essentially a fancy kind of string literal where the compiler checks that you have something of the given name, and Visual Studio knows what it refers to, so navigation and refactoring will work:

```
(if x == null) throw new ArgumentNullException(nameof(x));
```

You can put more elaborate dotted names in a nameof expression, but that's just to tell the compiler where to look: only the final identifier will be used:

```
WriteLine(nameof(Person.Address.ZipCode)); // prints "ZipCode"
```

9 Null-conditional operators

Sometimes code tends to drown a bit in null-checking. The null-conditional operator lets you access members and elements only when the receiver is not-null, providing a null result otherwise:

```
int? length = customers?.Length; // null if customers is null
Customer first = customers?[0];  // null if customers is null
```

The null-conditional operator is conveniently used together with the null coalescing operator ??:

```
int length = customers?.Length ?? 0; // 0 if customers is null
```

The null-conditional operator exhibits short-circuiting behavior, where an immediately following chain of member accesses, element accesses and invocations will only be executed if the original receiver was not null:

```
int? first = customers?[0].Orders.Count();
```

This example is essentially equivalent to:

```
int? first = (customers != null) ? customers[0].Orders.Count() : null;
```

Except that `customers` is only evaluated once. None of the member accesses, element accesses and invocations immediately following the ? are executed unless `customers` has a non-null value.

Of course null-conditional operators can themselves be chained, in case there is a need to check for null more than once in a chain:

```
int? first = customers?[0].Orders?.Count();
```

Note that an invocation (a parenthesized argument list) cannot immediately follow the ? operator – that would lead to too many syntactic ambiguities. Thus, the straightforward way of calling a delegate *only* if it's there does not work. However, you can do it via the `Invoke` method on the delegate:

```
if (predicate?.Invoke(e) ?? false) { … }
```

We expect that a very common use of this pattern will be for triggering of events:

```
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(nameof(X)));
```

This is an easy and thread-safe way to check for null before you trigger an event. The reason it's thread-safe is that the feature evaluates the left-hand side only once, and keeps it in a temporary variable.

10 Index initializers

Note: Index initializers have been previously previewed but do not work in the current CTP.

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 and other objects with indexers 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"
};
```

11 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.

12 Binary literals and digit separators

Note: These features are not implemented in C# in the current CTP.

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.

13 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.

14 Improved overload resolution

There are a number of small improvements to overload resolution, which will likely result in more things just working the way you'd expect them to. The improvements all relate to "betterness" – the way the compiler decides which of two overloads is better for a given argument.

One place where you might notice this (or rather stop noticing a problem!) is when choosing between overloads taking nullable value types. Another is when passing method groups (as opposed to lambdas) to overloads expecting delegates. The details aren't worth expanding on here – just wanted to let you know!