

Welcome to C# 9.0



Mads

May 20th, 2020



C# 9.0 is taking shape, and I'd like to share our thinking on some of the major features we're adding to this next version of the language.

With every new version of C# we strive for greater clarity and simplicity in common coding scenarios, and C# 9.0 is no exception. One particular focus this time is supporting terse and immutable representation of data shapes.

Let's dive in!

Init-only properties

Object initializers are pretty awesome. They give the client of a type a very flexible and readable format for creating an object, and they are especially great for nested object creation where a whole tree of objects is created in one go. Here's a simple one:

```
new Person
{
    FirstName = "Scott",
    LastName = "Hunter"
}
```

Object initializers also free the type author from writing a lot of construction boilerplate – all they have to do is write some properties!

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

The one big limitation today is that the properties have to be *mutable* for object initializers to work: They function by first calling the object's constructor (the default, parameterless one in this case) and then assigning to the property setters.

Init-only properties fix that! They introduce an `init` accessor that is a variant of the `set` accessor which can only be called during object initialization:

```
public class Person
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
}
```

With this declaration, the client code above is still legal, but any subsequent assignment to the `FirstName` and `LastName` properties is an error.

Init accessors and readonly fields

Because `init` accessors can only be called during initialization, they are allowed to

mutate **readonly** fields of the enclosing class, just like you can in a constructor.

```
public class Person
{
    private readonly string firstName;
    private readonly string lastName;

    public string FirstName
    {
        get => firstName;
        init => firstName = (value ?? throw new
ArgumentNullException(nameof(FirstName)));
    }
    public string LastName
    {
        get => lastName;
        init => lastName = (value ?? throw new
ArgumentNullException(nameof(LastName)));
    }
}
```

Records

Init-only properties are great if you want to make individual properties immutable. If you want the whole object to be immutable and behave like a value, then you should consider declaring it as a *record*:

```
public data class Person
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
}
```

The **data** keyword on the class declaration marks it as a record. This imbues it with several additional value-like behaviors, which we'll dig into in the following. Generally speaking, records are meant to be seen more as "values" – data! – and less as objects. They aren't meant to have mutable encapsulated state. Instead you represent change over time by creating new records representing the new state. They are defined not by their identity, but by their contents.

With-expressions

When working with immutable data, a common pattern is to create new values from existing ones to represent a new state. For instance, if our person were to change their last name we would represent it as a new object that's a copy of the old one, except with a different last name. This technique is often referred to as *non-destructive mutation*. Instead of representing the person *over time*, the record represents the person's state *at a given time*.

To help with this style of programming, records allow for a new kind of expression; the **with**-expression:

```
var otherPerson = person with { LastName = "Hanselman" };
```

With-expressions use object initializer syntax to state what's different in the new object from the old object. You can specify multiple properties.

A record implicitly defines a **protected** "copy constructor" – a constructor that takes an existing record object and copies it field by field to the new one:

```
protected Person(Person original) { /* copy all the fields */ } // generated
```

The **with** expression causes the copy constructor to get called, and then applies the object initializer on top to change the properties accordingly.

If you don't like the default behavior of the generated copy constructor you can define your own instead, and that will be picked up by the `with` expression.

Value-based equality

All objects inherit a virtual `Equals(object)` method from the `object` class. This is used as the basis for the `Object.Equals(object, object)` static method when both parameters are non-null.

Structs override this to have "value-based equality", comparing each field of the struct by calling `Equals` on them recursively. Records do the same.

This means that in accordance with their "value-ness" two record objects can be equal to one another without being the *same* object. For instance if we modify the last name of the modified person back again:

```
var originalPerson = otherPerson with { LastName = "Hunter" };
```

We would now have `ReferenceEquals(person, originalPerson) = false` (they aren't the same object) but `Equals(person, originalPerson) = true` (they have the same value).

If you don't like the default field-by-field comparison behavior of the generated `Equals` override, you can write your own instead. You just need to be careful that you understand how value-based equality works in records, especially when inheritance is involved, which we'll come back to below.

Along with the value-based `Equals` there's also a value-based `GetHashCode()` override to go along with it.

Data members

Records are overwhelmingly intended to be immutable, with init-only public properties that can be non-destructively modified through `with`-expressions. In order to optimize for that common case, records change the defaults of what a simple member declaration of the form `string FirstName` means. Instead of an implicitly private field, as in other class and struct declarations, in records this is taken to be shorthand for a public, init-only auto-property! Thus, the declaration:

```
public data class Person { string FirstName; string LastName; }
```

Means exactly the same as the one we had before:

```
public data class Person
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
}
```

We think this makes for beautiful and clear record declarations. If you really want a private field, you can just add the `private` modifier explicitly:

```
private string firstName;
```

Positional records

Sometimes it's useful to have a more positional approach to a record, where its contents are given via constructor arguments, and can be extracted with positional deconstruction.

It's perfectly possible to specify your own constructor and deconstructor in a record:

```
public data class Person
{
    string FirstName;
    string LastName;
    public Person(string firstName, string lastName)
        => (FirstName, LastName) = (firstName, lastName);
    public void Deconstruct(out string firstName, out string lastName)
        => (firstName, lastName) = (FirstName, LastName);
}
```

But there's a much shorter syntax for expressing exactly the same thing (modulo casing of parameter names):

```
public data class Person(string FirstName, string LastName);
```

This declares the public init-only auto-properties *and* the constructor *and* the deconstructor, so that you can write:

```
var person = new Person("Scott", "Hunter"); // positional construction
var (f, l) = person;                        // positional deconstruction
```

If you don't like the generated auto-property you can define your own property of the same name instead, and the generated constructor and deconstructor will just use that one.

Records and mutation

The value-based semantics of a record don't gel well with mutable state. Imagine putting a record object into a dictionary. Finding it again depends on `Equals` and (sometimes) `GetHashCode`. But if the record changes its state, it will also change what it's equal to! We might not be able to find it again! In a hash table implementation it might even corrupt the data structure, since placement is based on the hash code it has "on arrival"!

There are probably some valid advanced uses of mutable state inside of records, notably for caching. But the manual work involved in overriding the default behaviors to ignore such state is likely to be considerable.

With-expressions and inheritance

Value-based equality and non-destructive mutation are notoriously challenging when combined with inheritance. Let's add a derived record class `Student` to our running example:

```
public data class Person { string FirstName; string LastName; }
public data class Student : Person { int ID; }
```

And let's start our `with`-expression example by actually creating a `Student`, but storing it in a `Person` variable:

```
Person person = new Student { FirstName = "Scott", LastName = "Hunter", ID =
    GetNewId() };
otherPerson = person with { LastName = "Hanselman" };
```

At the point of that `with`-expression on the last line the compiler has no idea that `person` actually contains a `Student`. Yet, the new person wouldn't be a proper copy if it wasn't *actually* a `Student` object, complete with the same `ID` as the first one copied over.

C# makes this work. Records have a hidden virtual method that is entrusted with "cloning" the *whole* object. Every derived record type overrides this method to call the copy constructor of that type, and the copy constructor of a derived record chains to the copy constructor of the base record. A `with` expression simply calls the hidden

the copy constructor of the base record. A `with`-expression simply calls the hidden “clone” method and applies the object initializer to the result.

Value-based equality and inheritance

Similarly to the `with`-expression support, value-based equality also has to be “virtual”, in the sense that `Students` need to compare all the `Student` fields, even if the statically known type at the point of comparison is a base type like `Person`. That is easily achieved by overriding the already virtual `Equals` method.

However, there is an additional challenge with equality: What if you compare two *different* kinds of `Person`? We can’t really just let one of them decide which equality to apply: Equality is supposed to be symmetric, so the result should be the same regardless of which of the two objects come first. In other words, they have to *agree* on the equality being applied!

An example to illustrate the problem:

```
Person person1 = new Person { FirstName = "Scott", LastName = "Hunter" };
Person person2 = new Student { FirstName = "Scott", LastName = "Hunter", ID =
GetNewId() };
```

Are the two objects equal to one another? `person1` might think so, since `person2` has all the `Person` things right, but `person2` would beg to differ! We need to make sure that they both agree that they are different objects.

Once again, C# takes care of this for you automatically. The way it’s done is that records have a virtual protected property called `EqualityContract`. Every derived record overrides it, and in order to compare equal, the two objects must have the same `EqualityContract`.

Top-level programs

Writing a simple program in C# requires a remarkable amount of boilerplate code:

```
using System;
class Program
{
    static void Main()
    {
        Console.WriteLine("Hello World!");
    }
}
```

This is not only overwhelming for language beginners, but clutters up the code and adds levels of indentation.

In C# 9.0 you can just choose to write your main program at the top level instead:

```
using System;

Console.WriteLine("Hello World!");
```

Any statement is allowed. The program has to occur after the `usings` and before any type or namespace declarations in the file, and you can only do this in one file, just as you can have only one `Main` method today.

If you want to return a status code you can do that. If you want to `await` things you can do that. And if you want to access command line arguments, `args` is available as a “magic” parameter.

Local functions are a form of statement and are also allowed in the top level program. It is an error to call them from anywhere outside of the top level statement section.

Improved pattern matching

Several new kinds of patterns have been added in C# 9.0. Let's look at them in the context of this code snippet from the [pattern matching tutorial](#):

```
public static decimal CalculateToll(object vehicle) =>
    vehicle switch
    {
        ...

        DeliveryTruck t when t.GrossWeightClass > 5000 => 10.00m + 5.00m,
        DeliveryTruck t when t.GrossWeightClass < 3000 => 10.00m - 2.00m,
        DeliveryTruck _ => 10.00m,

        _ => throw new ArgumentException("Not a known vehicle type",
            nameof(vehicle))
    };

```

Simple type patterns

Currently, a type pattern needs to declare an identifier when the type matches – even if that identifier is a discard `_` as in `DeliveryTruck _` above. But now you can just write the type:

```
DeliveryTruck => 10.00m,

```

Relational patterns

C# 9.0 introduces patterns corresponding to the relational operators `<`, `<=` and so on. So you can now write the `DeliveryTruck` part of the above pattern as a nested switch expression:

```
DeliveryTruck t when t.GrossWeightClass switch
{
    > 5000 => 10.00m + 5.00m,
    < 3000 => 10.00m - 2.00m,
    _ => 10.00m,
},

```

Here `> 5000` and `< 3000` are relational patterns.

Logical patterns

Finally you can combine patterns with logical operators `and`, `or` and `not`, spelled out as words to avoid confusion with the operators used in expressions. For instance, the cases of the nested switch above could be put into ascending order like this:

```
DeliveryTruck t when t.GrossWeightClass switch
{
    < 3000 => 10.00m - 2.00m,
    >= 3000 and <= 5000 => 10.00m,
    > 5000 => 10.00m + 5.00m,
},

```

The middle case there uses `and` to combine two relational patterns and form a pattern representing an interval.

A common use of the `not` pattern will be applying it to the `null` constant pattern, as in `not null`. For instance we can split the handling of unknown cases depending on whether they are null:

```
not null => throw new ArgumentException($"Not a known vehicle type:
{vehicle}", nameof(vehicle)),
null => throw new ArgumentNullException(nameof(vehicle))

```

Also `not` is going to be convenient in if-conditions containing is-expressions where, instead of unwieldy double parentheses:


```
if (!(e is Customer)) { ... }
```

You can just say

```
if (e is not Customer) { ... }
```

Improved target typing

“Target typing” is a term we use for when an expression gets its type from the context of where it’s being used. For instance `null` and lambda expressions are always target typed.

In C# 9.0 some expressions that weren’t previously target typed become able to be guided by their context.

Target-typed `new` expressions

`new` expressions in C# have always required a type to be specified (except for implicitly typed array expressions). Now you can leave out the type if there’s a clear type that the expressions is being assigned to.

```
Point p = new (3, 5);
```

Target typed `??` and `?:`

Sometimes conditional `??` and `?:` expressions don’t have an obvious shared type between the branches. Such cases fail today, but C# 9.0 will allow them if there’s a target type that both branches convert to:

```
Person person = student ?? customer; // Shared base type  
int? result = b ? 0 : null; // nullable value type
```

Covariant returns

It’s sometimes useful to express that a method override in a derived class has a more specific return type than the declaration in the base type. C# 9.0 allows that:

```
abstract class Animal  
{  
    public abstract Food GetFood();  
    ...  
}  
class Tiger : Animal  
{  
    public override Meat GetFood() => ...;  
}
```

And much more...

The best place to check out the full set of upcoming features for C# 9.0 and follow their completion is the [Language Feature Status](#) on the Roslyn (C#/VB Compiler) GitHub repo.

Happy Hacking!



[Mads Torgersen](#)

C# Lead Designer, .NET Team

Follow



Posted in .NET

Read next

Using Visual Studio Codespaces with .NET Core

Learn about how we are enabling .NET Core projects for Codespaces when using Visual Studio 2019.

 **Tim Heuer** May 21, 2020

 **13 comments**

Introducing YARP Preview 1

YARP is an extensible open source reverse proxy using .NET. The Preview 1 library is now available on nuget.

 **Sam Spencer** May 21, 2020

 **25 comments**

277 comments

Comments are closed. [Login to edit/delete your existing comments](#)

[Newer Comments →](#)



Dominik Jeske May 20, 2020 8:29 am



What about source generators? There is no info in article and language status page. So it is planned in c# 9 or not?




Steve May 20, 2020 8:58 am



This is all excellent!

Will record types work in .NET Std 2.0 dlls? I ask as I have to make my shared code available to UWP.



Andy Gocke  May 21, 2020 10:58 am



Hey Steve, the current plan is that C# 9 will only be officially supported on .NET 5. However, in the current design records do not depend on any new CLR features, so many things may work in .NET Standard 2.0 in an unsupported fashion (similar to C# 8).



devtommy May 29, 2020 9:12 am



Why you will not support C#9 and C# 8 in UWP ???



Gavin Williams June 3, 2020 4:43 pm



@devtommy – you can actually use C#8 (mostly) in UWP today by adding a file at the solution level in Solution Explorer, calling it Directory.Build.Props and edit it's contents as follows ...


```
// ffs I can't even past some xml here. Put angle brackets at the
start of these lines
```

```
Project>
PropertyGroup>
  LangVersion>8.0
/PropertyGroup>
/Project>
```

You will want to move to .Net 5 when it's released at the end of the year, UWP isn't getting any new features as far as I can tell and is actually years behind in it's technology support now. All future development will likely go into .Net 5. MS is free to correct me if they disagree and are secretly holding back on UWP updates.



Michael Taylor May 20, 2020 9:01 am



Please provide links for each of these new features to the appropriate place in Github to provide feedback on them. In some cases (e.g. records) the issue is available by going to the list of new features. But other things (e.g. init only properties) don't seem to have an entry in the list and searching returns too many results.



Mads Torgersen  May 21, 2020 10:46 am



Yeah, sorry, init accessors have gotten lumped in with the records work. Here's the proposal for [init-only properties](#).



MgSam May 20, 2020 9:49 am



Mads, you are burying the lead for target-typed **new** expressions. I was originally opposed to this feature because it doesn't at first glance seem to add any new expressiveness, but it is fantastic for declaring and initializing collections. That is what you guys should be showing off.

```
var a = new[]
{
    new Foo(1, 2, 3),
    new Foo(1, 2, 3),
    new Foo(1, 2, 3)
}
```

vs

```
var a = new Foo[]
{
    new (1, 2, 3),
    new (1, 2, 3),
    new (1, 2, 3)
}
```



Andrew Hanlon May 20, 2020 1:38 pm



While likely worse in overhead and effort, there has been a way to have cleaner collection init syntax available in C# for several versions, namely extension **Add** methods:

```
public static class FooExt
{
    public static void Add(this IList<Foo> list, int a, int b, int c) => list.Add(new Foo(a, b,
c));
}

var foos = new List<Foo>
{
    {1,2,3},
    {1,2,3},
    {1,2,3}
};
```



MgSam May 20, 2020 2:00 pm



This wouldn't work for an array.



Andrew Hanlon May 20, 2020 2:05 pm



`.ToArray();`
But that is why I mentioned overhead etc. – I am still a fan of the new syntax.



Dominic Marius May 21, 2020 5:51 pm



Looking at the Target-Typed new expression feature,
Should this not be added.

```
class Person (p1,..pnth)

Person[] person = {{p1,...pnth},{..}} //Target type is explicitly specified already
```



Mads Torgersen May 21, 2020 10:39 am



Good point. I started mentioning this in my Build sessions after I saw your comment.



Alexander Omelchuk May 20, 2020 10:11 am



Best announcement of Build 2020. A big thanks to the team!



Zhao Wei Yi May 20, 2020 4:59 pm



I believe that less is better for a language.



Juan Manuel Barahona May 20, 2020 10:17 am



Awesome! thanks for these excellent features, but what about the performance of records?



Mads Torgersen May 21, 2020 10:49 am



It's as you'd expect, in the sense that there are no hidden or unexpected costs. Supporting non-destructive mutation means copying the object field by field. Value-based equality means comparing the object field by field, sometimes recursively if the fields also contain records with value-based equality. It's good to be aware of these costs as part of your decision to use records.



Hannes Kochniß May 21, 2020 12:50 pm



Ok, so the JIT doesn't understand records yet (as it once didn't care about sealed classes or prob. still doesn't understand the Index class), but might there be a future where the immutability of records is taken advantage of in the runtime? Thinking similar to readonly static fields/sealed class/readonly structs etc.



Nicolas Trahan May 25, 2020 1:06 pm



Yes, I wonder if structural sharing with copy on write could be implemented for records (see for instance Immer in the React/Redux world).



x x May 20, 2020 10:43 am



So you want to introduce "records" but you name them "data classes." Yeah that's logical... Why not just "record" keyword then? Because it is too Pascal-esque?



Cédric POTE May 20, 2020 12:49 pm



I suppose it's because making the word "record" become a keyword would prevent some legitimately named "record" variables.
Whereas preventing naming a variable "data" is much more acceptable.
Just my guess.



Jonathan LEI May 20, 2020 12:54 pm



Definitely not the reason. They can make it a contextual keyword such that it's not made reserved. This is exactly what they're doing to *data*.

Also, how would *data* be used less as identifier compared to *record* anyways.



Andrey Kontorovich May 20, 2020 2:02 pm



I think the reason is just simplicity. When you see "data class" you understand, that it's "class" with specific semantics for some cases. But it still behaves like class in terms of memory location / generics / etc. With word "record" you have to mention somewhere else that it's still class, but init only and with overridden equality members.



Darren Woodford May 30, 2020 3:42 pm



You could just as easily say "record class"



Todd Patrick Marek June 16, 2020 5:35 am



Why not use "case class" for immutable data classes as is done in Scala.

<https://docs.scala-lang.org/tour/case-classes.html>



Nicholas Bauer July 1, 2020 6:26 am



I have no idea what case would mean, having no Scala exposure. I can tell you what data means and I know the concept of a record generally.



Şafak Gür May 21, 2020 5:52 am



They may have wanted to keep the door open for “data structs”.



Daniel Earwicker May 21, 2020 10:27 am



Haskell example:

```
data Person = Person { firstName :: String, lastName :: String }
```



Mads Torgersen May 21, 2020 10:53 am



The current plan is to use “record” as the keyword, but our prototype still says “data” based on earlier proposals. We are still considering whether you should say “record class” or just “record”. It hinges on whether you also need struct records to be a thing in the language. We suspect we might get away with making *all* structs records – after all they already have field based copying (on assignment), so we could trivially implement with-expressions for them. And they also have value-based equality. If there’s no need to ever say “struct record” then “record” could always just *mean* “record class”.



Christian Brevik May 24, 2020 5:20 am



Since the behavior of Records is so distinct from normal classes, it seems like a good idea to drop the “class” keyword? Value-based equality, the default access modifier is public, etc.

My first reaction was that it seems like a potential footgun to use “class”.



Karl von Laudermann June 1, 2020 8:09 am



Personally, I would prefer to go the other way: keep “data” as the keyword, and drop the use of the term “record” entirely, referring to the feature as “data classes” instead. “Record” has baggage from other languages, such as Pascal. “Data class” makes it clear that it’s a class, just like any other, but set up to be suited for use as a collection of immutable data values.



Gavin Williams June 3, 2020 4:53 pm



Yep, I like data as well. Record means absolutely nothing to me. We understand data.



monkey noises June 15, 2020 6:45 am



How about:

immut class



George Birbilis June 24, 2020 5:46 pm



"We suspect we might get away with making all structs records"

Aren't structs value types though?

Records seem to be reference types instead, although at first I thought they'd be value types like structs, but behaving more like classes (just got an answer though on other thread [on stack-based allocation for classes] that they're reference types:

<https://github.com/dotnet/csharpplang/issues/3166#issuecomment-648831123>)

stack vs heap allocation is important as a difference I guess (for some apps it may be critical)



Andy Hames July 1, 2020 3:54 am



I think it's definitely better to drop the "class", whether you use "data" or "record". I think "record" also makes more sense, in my mind it infers more of a sense of immutability than "data". But then, "record" to me means the plastic spinny things first and foremost!



Michael Winking July 12, 2020 1:25 am



Have you considered implementing this without any new keywords? This could be done by starting from the short form.

First note that short form would already be non-ambiguous even without data due to the parenthesis after the class name so it shouldn't be too much of a problem for the parser to distinguish the following from a regular non-record definition.

```
public class Point(double X, double Y);
```

Second step would be to allow mixed short and long forms, like:

```
public class NamedPoint(double X, double Y)
{
    public string Name { get; init; }
}
```

The pure long form could then concisely be indicated by just using empty parenthesis as follows (with inheritance thrown in for good measure):

```
public class NamedPoint() : Point
{
    public string Name { get; init; }
}
```

On a second note, since C# now has records and pattern matching a further logical improvement would be to add some form of discriminated unions to allow for exhaustiveness checking (as in F#). It shouldn't be too hard to add this with some slight addition to the above proposal, i.e. something like the following might work:

```
public class Geom = Point(double X, double Y) { ...optional... }
    , Circle(double X, double Y, double Diameter) { ...optional... };
```

That doesn't look too terrible. Here the equals sign would allow the parser to distinguish it from the single record definition. The idea being that all the cases below would implicitly inherit from abstract base class Geom (it might make sense to allow Circle to derive from Point). In addition the compiler would seal everything so that no derived classes could be added (which would break exhaustiveness checking).



Marek Ištvanek June 3, 2020 7:37 am



I would prefer using `readonly class` instead of `data class`.



Andy Hames July 1, 2020 3:58 am



This is also not a bad shout.



Nicholas Bauer July 1, 2020 6:31 am



That seems like a fairly natural extension.



navnath kale May 20, 2020 10:44 am



Excellent !!! I have been craving for Covariant returns feature. Finally i can tidy up some of my old code. Rest of goodness is okay not really stopping dev from design decisions.

How about generic type inference at class level #3465 ? When can we expect that ?



Moien Tajik May 20, 2020 11:29 am



A big thanks to the team, you're awesome!



Wil Wilder Apaza Bustamante July 13, 2020 7:55 am



+1 , thank you!



Jimmy Choca May 20, 2020 11:52 am



This is awesome.

Any thoughts on having init-only required properties? For example,

```
public class Person
{
    public string FirstName { get; required; }
    public string LastName { get; required; }
}
```



MgSam May 20, 2020 12:03 pm



They are separately thinking about some kind of "validator" feature that would allow you to define validation logic for a class that would run after init-only properties are assigned. I doubt it'll be in the C# 9.0 release though.



Jimmy Choca May 20, 2020 1:50 pm



That would be cool



Mark Murphy May 20, 2020 6:11 pm



But for required init-only properties don't we want the ability to use non-nullable types for our properties....? Don't we need some way to tell the compiler "this property must be included in the object initializer so it doesn't

need a default value”? Otherwise aren’t we essentially forced to make e.g. all required string properties nullable...? I’m not sure value checks in the “validator” will solve this issue....

Edit: please also note we may want a value type to be a required init-only properties too as its default value might be meaningless....



Константин Салтук

May 20, 2020 11:41 pm



Actually it should.
For example, Resharper introduced nullability long time ago, using [NotNull], [CanBeNull], [ItemNotNull], etc.
And it will prevent you from creating instance of class with [NotNull] field without initializing this field with corresponding warning.
I think standard compiler nullability checking will warn you in case of creating record type with not nullable property without initializing it.



Stilgar Naib

May 21, 2020 3:14 am



I really hope the current implementation of records will indeed produce nullability warning if non-nullable properties are not assigned. Otherwise what was the point of the nullability feature in C# 8.0?



Daniel Earwicker

May 21, 2020 10:29 am



Could be an invariant that was checkable at other stages, e.g. in a debug build before/after every method call.



Mads Torgersen



May 21, 2020 11:01 am



We’ve gotten mixed feedback on those – we call them “final initializers” now – where some folks are worried about the complexity of ensuring that they always get called, even in reflection/generic/dynamic scenarios. I still really like the feature and I hope we can land it.



Peter Dolkens

June 29, 2020 4:50 pm



Even in your example, you’re effectively writing the boilerplate for

```
public string FirstName { get; required; }
```

I think it would be a massive miss for this feature to be missing what should be a fairly easy pre-processor step.

I loathe adding constructors with 50 parameters, when all I’m going to do with those parameters is load them into the public properties anyways.

Please put the effort in, this feature is probably the one I’m 2nd most excited about, and I know I’ll end up writing the boilerplate myself if I have to, but the potential is right there!



Phil Martin

May 20, 2020 5:20 pm



Yes, this was the first thing that came point mind. I like properties because they are a more descriptive way of constructing, and would very much like to have the compiler enforce certain properties to be initialized on construction.



[Mads Torgersen](#)  May 21, 2020 10:59 am



We thought a bit about required properties, but it’s complicated. It’s one of a couple of things that you get with constructor parameters but not object initializers. We’ll keep thinking about it.



[Mark Adamson](#) May 21, 2020 12:38 pm



If you don’t apply the nullability check on unset properties at compile time now for record initialisers then presumably it won’t be able to be added later. It would break any code that was previously not initialising the non-nullable properties.

I hope you are able to solve at least that case

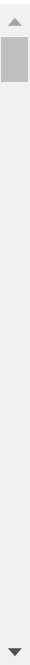
[Newer Comments →](#)

Relevant Links

- .NET Download
- .NET Hello World
- .NET Meetup Events
- .NET Documentation
- .NET API Browser
- .NET SDKs
- .NET Application Architecture Guides**
- Web apps with ASP.NET Core
- Mobile apps with Xamarin.Forms
- Microservices with Docker Containers
- Modernizing existing .NET apps to the cloud

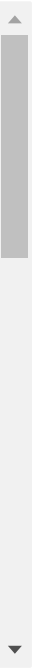
Archive

- October 2020
- September 2020
- August 2020
- July 2020
- June 2020
- May 2020
- April 2020
- March 2020
- February 2020
- January 2020
- December 2019



Topics

- Dot.Net
- .NET
- .NET Core
- .NET Framework
- Entity Framework
- C#
- ML.NET
- Visual Studio
- F#
- WPF
- Machine Learning



Stay informed



What's new

- Surface Duo
- Surface Laptop Go
- Surface Pro X
- Surface Go 2
- Surface Book 3
- Microsoft 365
- Windows 10 apps

Microsoft Store

- Account profile
- Download Center
- Microsoft Store support
- Returns
- Order tracking
- Virtual workshops and training
- Microsoft Store Promise

Education

- Microsoft in education
- Office for students
- Office 365 for schools
- Deals for students & parents
- Microsoft Azure in education

Enterprise

- Azure
- AppSource
- Automotive
- Government
- Healthcare
- Manufacturing
- Financial services
- Retail

Developer

- Microsoft Visual Studio
- Windows Dev Center
- Developer Center
- Microsoft developer program
- Channel 9
- Office Dev Center
- Microsoft Garage

Company

- Careers
- About Microsoft
- Company news
- Privacy at Microsoft
- Investors
- Diversity and inclusion
- Accessibility
- Security