Covers C# 6 and 7

C# IN DEPTH

FOURTH EDITION

MEAP

Jon Skeet

MANNING

MEAP Edition
Manning Early Access Program
C# in Depth
Fourth Edition
Covers C# 6 and 7
Version 13

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *C# in Depth, 4th edition*. The book is currently still very much a work in progress, but I'm excited to share it with you nonetheless – partly as it's always nice to share knowledge, and partly so that I can incorporate your feedback into the chapters presented here and ones that have yet to be written. Whether you're new to the series or whether this is the first time you've picked up the book, your feedback is very valuable to me.

As a language, C# has been getting better and better – and with .NET Core going cross-platform and so many components being developed in the open, there's never been a better time to be a .NET developer. Like all languages, the better you know C# the more effective you'll be with it.

Since the first edition, my aim with *C# in Depth* has been to hone existing C# developers, going deeper into how language features work, why they were designed that way, where they have limitations, and the best uses for them. If you're very new to C#, this may not be the book for you – but if you already have some experience, I hope you'll find this will deepen your relationship with the language.

We're starting to get pretty close to a complete book at this stage. C# 2 and 3 are "in the can" so to speak, and I'm about to start writing about C# 4. At the same time, I've recently been learning a lot more about what C# 8 has in store for us. Additionally, C# 7.3 has now been released, so I can check the changes I'd already written about, and add a few additional ones. That means there are still a few changes to the later chapters, and of course I still need to write chapter 1. But the end is definitely in sight!

Do check out the Author Online forum, and let me know how you find this early access version – what works for you as well as anything that didn't quite make sense. With your help, we can make this fourth edition the best yet!

—Jon Skeet

brief contents

1

Survival of the sharpest

In this chapter you will learn:

- How C# has evolved rapidly in many directions to make developers more productive
- How to select minor versions of C# to use the latest features
- How C# can run in more and more environments
- How an open and engaged community helps everyone
- How this book works at different detail levels for old and new versions

It was hard to pick out the most interesting aspects of C# to introduce here. Some are fascinating but rarely used. Others are incredibly important but commonplace to C# developers now. Then there are features like async/await which are great in so many ways, but hard to describe briefly.

Without further ado, let's look at how far C# has come over time.

1.1 An evolving language

In previous editions of this book, I provided a single example that showed the evolution of the language over the versions covered by that edition. That's no longer feasible in a way that would be interesting to read. While a large application may use almost all of the new features, any single piece of code that's suitable for the printed page would only use a subset of them.

Instead, in this section I'll pick out what I consider to be the most important themes of C# evolution and give very brief examples of some of the improvements. This is far from an exhaustive list of features. It's also not intended to teach you the features as such. Instead, it's a reminder of how far we've come for features you already know about, and a tease for features you may not have seen yet.

If some of these features strike you as imitating other languages you're familiar with, you're almost certainly right. The C# team has no hesitation in taking great ideas from other

languages and reshaping them to feel at home within C#. This is a great thing! F# is particularly worth mentioning here as a source of inspiration for many C# features. It's possible that F#'s greatest impact is not what it enables for F# developers[1], but its influence on C#.

Let's start off with one of the most important aspects of C#: its type system.

1.1.1 A helpful type system at large and small scales

C# has been a statically-typed language from the start: your code specifies the types of variables, parameters, values returned from methods and so on. The more precisely you can specify the shape of the data your code accepts and returns, the more the compiler can help you avoid mistakes.

That's particularly true as the application you're building grows. If you can see all the code for your whole program on one screen (or at least hold it all in your head at one time) then a statically-typed language doesn't have much benefit. As the scale increases, it becomes increasingly important that your code communicates what it does concisely and effectively. You can do that through documentation, but static typing let's you communicate in a machine-readable way. As C# has evolved, its type system has allowed more fine-grained descriptions. The most obvious example of this is *generics*. In C# 1 we might have had code like this:

```
public class Bookshelf
{
    public IEnumerable Books { get { … } }
}
```

What type is each item in the `Books` sequence? The type system doesn't tell us. With generics in C# 2 we can communicate more effectively:

```
public class Bookshelf
{
    public IEnumerable<Book> Books { get { … } }
}
```

C# 2 also brought *nullable value types*, allowing the absence of information to be expressed effectively without resorting to magic values such as -1 for a collection index or `DateTime.MinValue` for a date.

C# 7 then gave us the ability to tell the compiler that a user-defined struct should be immutable, using `readonly struct` declarations. The primary goal for this feature may well have been to improve the efficiency of the code generated by the compiler, but it has additional benefits in terms of communicating intent.

The plans for C# 8 include *nullable reference types*, which will allow even more communication. Up to this point, there is nothing in the language to let us express whether a reference (either as a return value, or a parameter, or just a local variable) might be null or

[1] This is not to underplay the value of F# as a language in its own right, or to suggest that it shouldn't be used directly. But currently, the C# community is significantly larger than the F# community, and the C# community owes a debt of gratitude to F# for inspiring the C# team.

not. This leads to error-prone code if you're *not* careful, and boilerplate validation code if you *are* careful, neither of which is ideal. C# 8 will expect that anything which isn't explicitly nullable is intended not to be nullable. For example, consider a method declaration like this:

```
string Method(string x, string? y)
```

The parameter types indicate that the argument corresponding to `x` should not be null, but that the argument corresponding to `y` may be null. The return type indicates that the method won't return null.

Other changes to the type system in C# are aimed at a smaller scale, focusing on how one method might be implemented rather than between different components in a large system. C# 3 introduced both *anonymous types* and *implicitly-typed local variables* (`var`). These both help to address the downside of some statically-typed languages: verbosity. If you need a particular data shape within a single method but nowhere else, creating a whole extra type just for the sake of that method is overkill. Anonymous types allow that data shape to be expressed very concisely, without losing the benefits of static typing:

```
var book = new { Title = "Lost in the Snow", Author = "Holly Webb" };
string title = book.Title;         //#A
string author = book.Author;       //#A
```

#A: Name and type are still checked by the compiler

Anonymous types are primarily used within LINQ queries, but the principle of creating a type just for a single method doesn't depend on LINQ.

Similarly, specifying the type of a variable explicitly in a statement which also calls the constructor of that type seems redundant. I know which of the following declarations I find cleaner:

```
Dictionary<string, string> map1 = new Dictionary<string, string>(); //#A
var map2 = new Dictionary<string, string>();                        //#B
```

#A: Explicit typing
#B: Implicit typing

While implicit typing is most necessary when working with anonymous types, I've found it increasingly useful when working with regular types too. It's important to distinguish between *implicit* typing and *dynamic* typing. The `map2` variable above is still statically typed; it's just that we haven't had to write the type explicitly.

Anonymous types only help within a single block of code: you can't use them as method parameters or return types, for example. C# 7 introduced *tuples*; value types which effectively act to collect variables together. The framework support for these tuples is relatively simple, but there's additional language support to allow the elements of tuples to be named. For example, instead of the anonymous type above, we could use:

```
var book = (title: "Lost in the Snow", author: "Holly Webb");
Console.WriteLine(book.title);    //#A
```

Tuples can replace anonymous types in some cases, but certainly not all. One of their benefits is that they *can* be used as method parameters and return types. At the moment I would personally advise that these be kept within the internal API of a program rather than exposed publicly, as tuples represent a simple composition of values rather than encapsulating them. That's why I still regard them as contributing to the "code at a small scale" improvements of C#; I use them as an implementation detail.

I should mention *record types* at this point: a feature which *might* come in C# 8. I think of record types as "named anonymous types" to some extent, at least in their simplest form. They would provide the benefits of anonymous types in terms of removing boilerplate code, but then allow those types to gain extra behavior just like regular classes do. Watch this space!

1.1.2 Ever more concise code

One of the recurring themes within new features of C# has been the ability to let you express your ideas in a way that is more and more concise. The type system is part of this, as we've seen with anonymous types, but there are many other features that contribute to this. There are lots of words you might hear for this, especially in terms of what can be removed with the new features in place. C#'s features allow you to reduce ceremony, remove *boilerplate* code, and avoid cruft. These are just different ways of talking about the same effect. It's not that any of the now-redundant code was wrong; it was just distracting and unnecessary.

Let's look at a few of the ways that C# has evolved in this respect.

CONSTRUCTION AND INITIALIZATION

First, we'll consider how you create and initialize objects. Delegates have probably evolved the most, and in multiple stages. In C# 1, you had to write a separate method for the delegate to refer to, then create the delegate itself in a long-winded way. For example, here's what you'd write to subscribe a new event handler to a button's `Click` event in C# 1:

```
button.Click += new EventHandler(HandleButtonClick); //#A
```

#A: C# 1

C# 2 introduced both *method group conversions* and *anonymous methods*. If you wanted to keep the `HandleButtonClick` method, method group conversions would allow you to change the code above to:

```
button.Click += HandleButtonClick; //#A
```

#A: C# 2

If your click handler is simple, you might not want to bother with a separate method at all, and instead use an anonymous method:

```
button.Click += delegate { MessageBox.Show("Clicked!"); }; //#A
```

#A: C# 2

Anonymous methods have the additional benefit of acting as *closures*: they can use local variables in the context within which they're created. However, they're not used very often in modern C# code – because C# 3 provided us with *lambda expressions*, which have almost all the benefits of anonymous methods, but shorter syntax[2].

```
button.Click += (sender, args) => MessageBox.Show("Clicked!"); //#A
```

#A: C# 3

I used event handlers as an example for delegates because that was their main use in C# 1. In later versions of C#, delegates are used in more varied situations, particularly in LINQ.

LINQ also brought other benefits for initialization, in the form of *object initializers* and *collection initializers*. These allow you to specify a set of properties to set on a new object or items to add to a new collection, all within a single expression. It's simpler to show than describe, and I'll borrow an example from chapter 3. Consider code that you might previously have written like this:

```
var customer = new Customer();
customer.Name = "Jon";
customer.Address = "UK";
var item1 = new OrderItem();
item1.ItemId = "abcd123";
item1.Quantity = 1;
var item2 = new OrderItem();
item2.ItemId = "fghi456";
item2.Quantity = 2;
var order = new Order();
order.OrderId = "xyz";
order.Customer = customer;
order.Items.Add(item1);
order.Items.Add(item2);
```

The object and collection initializers introduced in C# 3 make this *so* much clearer:

```
var order = new Order
{
    OrderId = "xyz",
    Customer = new Customer { Name = "Jon", Address = "UK" },
    Items =
    {
        new OrderItem { ItemId = "abcd123", Quantity = 1 },
        new OrderItem { ItemId = "fghi456", Quantity = 2 }
    }
};
```

I wouldn't suggest reading either of these examples in detail; what's important is the simplicity of the second form over the first.

[2] Usually, anyway. In this case the lambda expression is longer than the anonymous method because the anonymous method uses the one feature lambda expressions don't have: the ability to ignore parameters by just not providing a parameter list.

METHOD AND PROPERTY DECLARATIONS

One of the most obvious examples of simplification is through *automatically implemented properties*. These were first introduced in C# 3, but have gained further improvements in later versions.

Consider a property which would have been implemented in C# 1 like this:

```
private string name;
public string Name
{
    get { return name; }
    set { name = value; }
}
```

Automatically implemented properties allow this to be written as a single line:

```
public string Name { get; set; }
```

Additionally, C# 6 introduced *expression-bodied members* which remove more ceremony. Suppose you're writing a class that wraps an existing collection of strings, and you want to effectively delegate the `Count` and `GetEnumerator()` members of your class to that collection. Prior to C# 6, you would have had to write something like this:

```
public int Count { get { return list.Count; } }

public IEnumerator<string> GetEnumerator()
{
    return list.GetEnumerator();
}
```

This is a strong example of ceremony: a lot of syntax that the language used to require, with very little benefit. In C# 6, this is significantly cleaner. The `=>` syntax (already used by lambda expressions) is used to indicate an expression-bodied member:

```
public int Count => list.Count;

public IEnumerator<string> GetEnumerator() => list.GetEnumerator();
```

While it's a personal and subjective matter, I've been surprised by just how much difference expression-bodied members have made to the readability of my code. I love them!

Another feature I hadn't expected to use as much as I now do is string interpolation – just one of the string-related improvements in C#.

STRING HANDLING

There have been three significant improvements around string handling in C#:

- C# 5 introduced *caller information attributes*, including the ability for the compiler to automatically populate method and file names as parameter values. This is great for diagnostic purposes, whether in permanent logging or more temporary testing.
- C# 6 introduced the `nameof` operator, allowing names of variables, types, methods and

other members to be represented in a refactoring-friendly form.
- C# 6 also introduced *interpolated string literals*. This is not a new concept, but makes constructing a string using dynamic values much simpler.

For the sake of brevity, I'll just demonstrate the last point. It's reasonably common to want to construct a string using variables, properties, the result of method calls and so forth. This might be for logging purposes, user-oriented error messages (if localization isn't required), exception messages and so forth.

Here's an example from my Noda Time project[3]. Users can try to find a calendar system by its ID, and the code throws a `KeyNotFoundException` if that ID doesn't exist. Prior to C# 6, the code might have looked like this:

```
throw new KeyNotFoundException(
    "No calendar system for ID "  + id + " exists");
```

Or using explicit string formatting:

```
throw new KeyNotFoundException(
    string.Format("No calendar system for ID {0} exists", id);
```

In C# 6, it's just a little simpler with an interpolated string literal to include the value of `id` in the string directly:

```
throw new KeyNotFoundException($"No calendar system for ID {id} exists");
```

This doesn't look like a big deal, but I'd hate to have to work without it now.

These are just the most prominent features that help to improve the signal-to-noise ratio of your code. I could have shown `using static` directives and the null conditional operator in C# 6, as well as pattern matching, deconstruction and out variables in C# 7. Rather than expand chapter 1 to mention every feature in every version, let's move on to a feature which is more revolutionary than evolutionary: LINQ.

1.1.3 Simple data access with LINQ

If you ask a C# developer what they love about C#, it's very likely to include LINQ. We've already seen some of the features which build up to LINQ, but the most radical is query expressions. Consider this code:

```
var offers =
    from product in db.Products
    where product.SalePrice <= product.Price / 2
    orderby product.SalePrice
    select new {
        product.Id, product.Description,
        product.SalePrice, product.Price
    };
```

[3] See section 1.4.2 for details of Noda Time, although you don't need to know about it to understand this example.

That just doesn't look anything like "old-school" C#. Imagine travelling back in time to 2007 to show that code to a developer using C# 2. Then go on to explain that this has compile-time checking and Intellisense support, and that it results in an efficient database query. Oh, and that you can use the same syntax for regular collections as well.

Support for querying out-of-process data is provided via *expression trees*. These represent code as data, and a LINQ provider can analyze the code to convert it into SQL or other query languages. While this is extremely cool, I rarely use it myself, simply because I don't work with SQL databases very often. I *do* work with in-memory collections though, and I use LINQ all the time, whether through query expressions or just method calls with lambda expressions.

LINQ didn't just give C# developers new tools: it encouraged us to think about data transformations in a new way, based on functional programming. This affects more than just data access. LINQ provided the initial impetus to take on more functional ideas, but many C# developers have embraced those ideas and taken them further.

C# 4 made a radical change in terms of dynamic typing, but I don't think that affected as many developers as LINQ. Then C# 5 came along and changed the game again, this time with respect to asynchrony.

1.1.4 Asynchrony

Asynchrony has been difficult in mainstream languages for a long time. More niche languages have been created with asynchrony in mind from the start, and some functional languages have made it relatively easy as just one of the things they handle neatly. But C# 5 brought a new level of clarity to programming asynchrony in a mainstream language, with a feature usually referred to as async/await.

The feature consists of two complementary parts, both around async methods[4]:

- Async methods *produce* a result representing an asynchronous operation with no effort on the part of the developer. This result type is usually `Task` or `Task<T>`.
- Async methods *consume* asynchronous operations using `await` expressions. If the method tries to await an operation which hasn't completed yet, the method "pauses asynchronously" until the operation completes, then continues.

The details of what's meant by "asynchronous operation" and "pausing asynchronously" are where things become tricky, and I'm not going to attempt to explain them now. But the upshot is that we can write code that is asynchronous, but mostly looks like the synchronous code we're more familiar with. It even allows for concurrency in a natural way.

As an example, consider this asynchronous method that might be called from a Windows Forms event handler:

```
private async Task UpdateStatus()
{
    Task<Weather> weatherTask = GetWeatherAsync();       //#A
```

[4] More properly asynchronous *functions*, as anonymous methods and lambda expressions can be asynchronous too.

```
    Task<EmailStatus> emailTask = GetEmailStatusAsync(); //#B

    Weather weather = await weatherTask;                 //#B
    EmailStatus email = await emailTask;                 //#B

    weatherLabel.Text = weather.Description;             //#C
    inboxLabel.Text = email.InboxCount.ToString();       //#C
}
```

#A: Start two operations concurrently
#B: Wait for them to complete, asynchronously
#C: Update the user interface

As well as starting two operations concurrently and then awaiting their results, this demonstrates how async/await is aware of synchronization contexts. We're updating the user interface, which can only be done in a UI thread, despite also starting and waiting for long-running operations. Before async/await, this would have been complex and error-prone.

I wouldn't claim async/await is a silver bullet for asynchrony. It doesn't magically remove all the complexity which naturally comes with the territory. Instead, it lets you focus on the inherently difficult aspects of asynchrony by taking away a lot of the boilerplate code that was previously required.

All of the features we've seen so far have aimed to make code simpler. The final aspect I would like to mention is slightly different.

1.1.5 Balancing efficiency and complexity

I remember my first experiences with Java. It was entirely interpreted, and painfully slow. After a while, optional JIT compilers became available, and eventually it was taken almost for granted than any Java implementation would be JIT-compiled.

Making Java perform well took a lot of effort. This effort would simply not have happened if the language had been a flop. But developers saw the potential, and already felt more productive than they had been before. Speed of development and delivery can often be more important than application speed.

C# was in a slightly different situation. The CLR has been "pretty good" in terms of efficiency right from the start. The language support for easy interop with native code and for performance-sensitive unsafe code with pointers both help too. C# performance still improved and continues to improve over time. (I note with a wry smile that Microsoft is now introducing tiered JIT compilation, broadly like the Java Hotspot JIT compiler.)

But different workloads have different performance demands. As we'll see in section 1.2, C# is now in use across a surprising variety of platforms, including gaming and microservices, both of which can have difficult performance requirements.

Asynchrony helps address performance in some situations, but C# 7 is the most obviously performance-sensitive release. Read-only structs and a much larger surface area for `ref` features both help to avoid redundant copying. The `Span<T>` feature present in modern frameworks and supported by ref-like struct types then reduces unnecessary allocation and garbage collection. The hope is clearly that when used carefully, these techniques will cater to the requirements of specific developers.

I have a slight sense of unease around these features, as they still feel complex to me. I can't reason about a method using an `in` parameter as clearly as I can around regular value parameters, and I'm sure it will take a while before I'm comfortable with what I can and can't do with ref locals and ref returns.

My hope is that these features will be used in moderation. They will simplify code in situations which really benefit from them, and will no doubt be welcomed by the developers who maintain that code. I look forward to experimenting with these features in personal projects and becoming more comfortable with the balance between improved performance and increased code complexity.

I don't want to sound this note of caution too loudly. I suspect the C# team made the right choice to include the new features regardless of how much or little *I* will use them in my work. I just want to point out that you don't have to use a feature just because it's there. Make your decision to opt into complexity a conscious one.

Speaking of opting in, C# 7 brought a new meta-feature to the table: the use of minor version numbers, for the first time since C# 1.

1.1.6 Evolution at speed: using minor versions

The set of version numbers for C# is an odd one, and complicated by the fact that many developers get understandably confused between the framework and the language. (There's no C# 3.5 for example. The .NET framework version 3.0 shipped with C# 2, and .NET 3.5 shipped with C# 3.) C# 1 had two releases: C# 1.0 and C# 1.2. Between C# 2 and C# 6 inclusive, there were just major versions, usually backed by a new version of Visual Studio.

C# 7 bucked that trend: we've had releases of C# 7.0, C# 7.1, C# 7.2 and C# 7.3, all available in Visual Studio 2017. I consider it highly likely that this pattern will continue in C# 8. The aim is to allow new features to evolve quickly with user feedback. The majority of C# 7.1-7.3 features have been tweaks or extensions to the features introduced in C# 7.0.

Volatility in language features can be disconcerting, particularly in large organizations. There may be a lot of infrastructure to change or upgrade in order to make sure the new language version is fully supported. There may be a lot of developers who learn and adopt new features at different paces. If nothing else, it can simply be a little uncomfortable for the language to change under your feet more often than you're used to.

For this reason, the C# compiler defaults to using the earliest minor version of the latest major version it supports. If you use a C# 7 compiler and don't specify any language version, it will restrict you to C# 7.0 by default. If you want to use a later minor version, you need to specify that in your project file explicitly, opting into the new features.

You can do this in two ways, although they have the same effect. You can edit your project file directly to add a `<LangVersion>` element in a `<PropertyGroup>`, like this:

```
<PropertyGroup>
  …                                    //#A
  <LangVersion>latest</LangVersion>    //#B
</PropertyGroup>
```

#A: Other properties
#B: Specify the language version of the project

If you don't like editing project files directly, you can go to the project properties in Visual Studio, select the Build tab, then click on the Advanced button in the bottom right. A dialog will appear to allow you to select the language version you wish to use, along with other options.

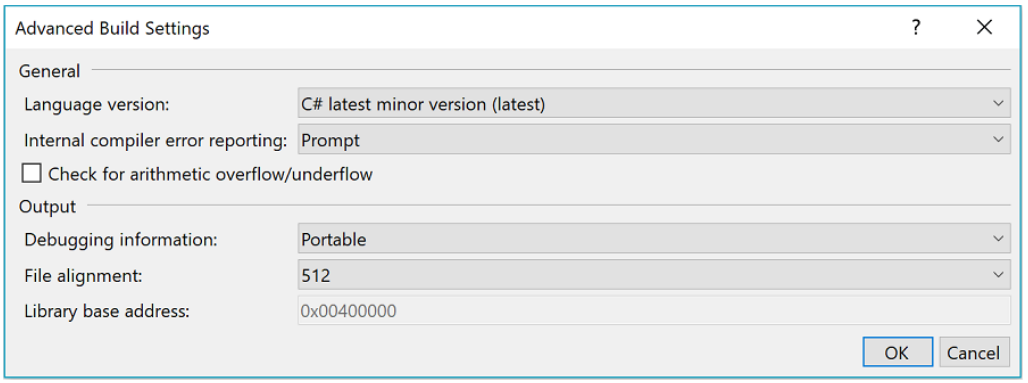

Figure 1.1 Language version settings in Visual Studio

This option in the dialog is not new, but you're more likely to want to use it now than in previous versions. The values you can select are:

- "default": The first release of the latest major version
- "latest": The latest version
- A specific version number, such as "7.0" or "7.3"

This doesn't change the version of the compiler you run. It changes the set of language features available to you. This means if you try to use something that isn't available in the version you're targeting, the compiler error message will usually explain which version is required for that feature. If you try to use a language feature that's entirely unknown to the compiler – using C# 7 features with a C# 6 compiler, for example – the error message is usually less clear.

So C# as a language has come a long way since its first release. What about the platform it runs on?

1.2 An evolving platform

The last few years have been exhilarating for .NET developers. There's been a certain amount of frustration as well, as both Microsoft and the .NET community come to terms with the implications of a more open development model. But the overall result of the hard work by so many people is quite remarkable.

For many years, running C# code would almost always mean running on Windows. It would usually mean either a client-side app written in Windows Forms or WPF, or a server-side app written with ASP.NET and probably running behind IIS. Other options have been available

for a long time, and the Mono project in particular has a rich history, but the mainstream of .NET development was still on Windows.

As I write this in June 2018, the .NET world is very different. The most prominent development is .NET Core: a runtime and framework which is both portable and open source, fully supported by Microsoft on multiple operating systems, complete with streamlined development tooling. Only a few years ago that would have been unthinkable. Add to that a portable and open source IDE in the form of Visual Studio Code, and you get a flourishing .NET ecosystem with developers working on all kinds of local platforms, and then deploying to all kinds of server platforms.

It would be a mistake to focus too heavily on .NET Core and ignore the multitude of other ways in which C# runs these days. Xamarin provides a rich multi-platform mobile experience. Its GUI framework (Xamarin Forms) allows developers to create user interfaces which are fairly uniform across different devices where that's appropriate, but which can take advantage of the underlying platform too.

Unity is one of the most popular game development platforms in the world. With a customized Mono runtime and ahead-of-time compilation, it can provide challenges to C# developers used to more traditional runtime environments. But for many developers, this is their first or perhaps *only* experience with the language.

These widely-adopted platforms are far from the only ones making C#. I've recently been working both Try .NET and Blazor, for very different forms of browser/C# interaction.

Try .NET allows users to write code in a browser, with auto-completion, and then build and run that code. It's *great* for experimenting with C# with a barrier to entry that's about as low as it can be.

Blazor is a platform for running Razor pages directly in a browser. These aren't pages rendered by a server and then displayed in the browser: the user interface code runs within the browser, using a version of the Mono runtime converted into WebAssembly. The idea of a whole runtime executing IL via the JavaScript engine in a browser, not just on full computers but also on mobile phones, would have struck me as absurd just a few years ago. I'm very glad other developers have more imagination.

A lot of the innovation in this space has only been made possible by a more collaborative and open community than ever before.

1.3 An evolving community

I've been involved in the C# community since the C# 1.0 days, and I've never seen it as vibrant as it is today. When I started using C#, it was very much seen as an "enterprise" programming language, and there was *relatively* little sense of fun and exploration[5]. With that background, the open source C# ecosystem grew fairly slowly compared with other languages – even Java, which was also "enterprisey". Around the time of C# 3, the alt.NET community

[5] Don't get me wrong, it was still a pleasant community to be part of, and there have always been people "playing" with C#.

was looking beyond the mainstream of .NET development, and this was seen as being "against" Microsoft in some senses.

In 2010, the NuGet (initially NuPack) package manager was launched[6], making it much easier to produce and consume class libraries, whether commercial or open source. Even though the barrier of downloading a zip file, copying a DLL into somewhere appropriate and then adding a reference to it doesn't *sound* hugely significant, every point of friction can put developers off.

Fast forward to 2014, and Microsoft announces that its Roslyn compiler platform is going to become open source, under the umbrella of the new .NET Foundation. Then .NET Core is announced, initially under the codename "Project K", with DNX coming later and finally the .NET Core tooling that's now released and stable. And ASP.NET Core. And Entity Framework Core. And Visual Studio Code. The list of products which truly live and breathe on GitHub goes on.

The technology has been important, but the new embrace of open source by Microsoft has been equally vital for a healthy community. Third-party open source packages have blossomed, including innovative uses for Roslyn and integrations within .NET Core tooling that "just feels right".

None of this has happened in a vacuum. The rise of Cloud Computing makes .NET Core even more important to the .NET ecosystem than it would have been otherwise; support for Linux isn't optional. But because .NET Core *is* available, there's now nothing special about packaging up an ASP.NET Core service in a Docker image, deploying it with Kubernetes, and using it as just one part of a larger application that could involve many different languages. The cross-pollination of good ideas between many communities has always been present, but is stronger than ever right now.

You can learn C# in a browser. You can run C# anywhere. You can ask questions about C# on Stack Overflow and myriad other sites. You can join in the discussion about the future of the language on the C# team's GitHub repository. It's not perfect: we still have collective work to do in order to make the C# community as welcoming as it possibly can be for *everyone*, but we're in a great place already.

I'd like to think that C# in Depth has its own small place in the C# community too. So how has this book evolved?

1.4 An evolving book

You're reading the fourth edition of C# in Depth. While the book hasn't evolved at the same pace as the language, platform or community, it's not the same book as it started out. This section will help you get the best out of the book.

[6] Other package managers were developed even earlier, and the OpenWrap project developed by Sebastien Lambla was particularly influential.

1.4.1 Mixed-level coverage

The first edition of C# in Depth came out in April 2008, coincidentally at the same time that I joined Google. Back then, I was aware that a lot of developers knew C# 1 fairly well but were picking up C# 2 and C# 3 as they went along, without a firm grasp of how all the pieces fit together. I aimed to address that gap, diving into the language at a depth that would help readers understand not just what each feature did, but why it was designed that way.

Over time, the needs of developers change. It feels to me that the community has absorbed a deeper understanding of the language almost by osmosis, at least for earlier versions of the language. This won't be a universal experience, but for the fourth edition I wanted the emphasis to be on the newer versions. I still think it's useful to understand the evolution of the language version-by-version[7], but there's less need to look at every detail of the features in C# 2-4.

I'm also not keen on very thick books. I don't want C# in Depth to be intimidating, or hard to hold, or hard to write on[8]. Keeping 400 pages of coverage for C# 2-4 just didn't feel right.

For that reason, I've compressed my coverage of those versions. Every feature is mentioned, and I go into detail where I feel it's appropriate, but there's less depth than the third edition contained. The good news is that buying the fourth edition gives you an electronic copy of the third edition as well. Use the coverage in the fourth edition as revision of topics you already know, and suggestions about where you might want to find out more from the third edition. Versions 5-7 of the language are covered in more detail. Asynchrony is still a tough topic to understand, and the third edition obviously doesn't cover C# 6 or 7 at all.

Writing, like software engineering, is often a balancing act. I hope the balance I've struck between detail and brevity works for you. Only time will tell what we'll do if a fifth edition ever comes into being.

1.4.2 Examples using Noda Time

Most of the examples I provide in the book are standalone. However, in order to make a more compelling case for some features, it's useful to be able to point to where I use them in production code. Most of the time, I use Noda Time for this.

Noda Time is an open source project I started in 2009 to provide a better date and time library for .NET. It's served a secondary purpose though: it's been a great "sandbox" project for me. It's helped me hone my API design skills, learn more about performance and benchmarking, and test out new C# features. All of this without breaking users, of course.

Every new version of C# has introduced features that I've been able to use in Noda Time, so I think it makes sense to use those as concrete examples in this book. All of the code is available on GitHub so you can clone it and experiment for yourself. The purpose of giving

[7] This isn't the best way to learn the language from scratch, but it's useful if you really want to understand it deeply. I wouldn't write a book for C# beginners using the same structure.

[8] If you've got a physical copy of this book, I strongly encourage you to write on it. Point out places where you disagree, or parts that are particularly useful. Just the act of doing this will reinforce it in your memory, and you'll have a reminder later.

examples using Noda Time is not to persuade you to use the library… but I'm not going to complain if that happens to be a side-effect.

For the rest of the book, I'll just assume that when I refer to Noda Time, you know what I'm talking about. In terms of making it suitable for examples, the important aspects of it are:

- The code needs to be as readable as possible. If a language feature lets me refactor for readability, I'll jump at the chance.
- Noda Time follows semantic versioning, and new major versions are rare. I pay attention to the backward compatibility aspects of applying new language features.
- While I don't have very concrete performance goals as Noda Time can be used in many different contexts, I pay attention to performance and will embrace features that improve efficiency, so long as they don't make the code much more complex.

To find out more about the project and check out its source code, visit https://nodatime.org or https://github.com/nodatime/nodatime.

1.4.3 Terminology choices

I've tried to follow the official C# terminology as closely as I can within the book, but occasionally I've allowed clarity to take precedence over precision. For example, when writing about asynchrony, I've often referred to "async methods" when the same information also applies to asynchronous anonymous functions. Likewise object initializers apply to accessible fields as well as properties, but it's simpler to mention that once and then only refer to properties within the rest of the explanation.

Sometimes the terms within the specification are rarely used in the wider community. For example, the specification has the notion of a "function member". That's a method, property, event, indexer, user-defined operator, instance constructor, static constructor or finalizer. It's a term for "any type member that can contain executable code" and it's useful when describing language features. It's not nearly as useful when you're looking at your own code, which is why you may never have heard of it before. I've tried to use terms like this sparingly, but my personal view is that it's worth becoming somewhat familiar with them, all in the spirit of getting closer to the language.

Finally, there are some concepts that don't have any official terminology, but which are still useful to refer to in a short-hand form. The one I'll use most often is probably *unspeakable names*. This term, coined by Eric Lippert[9], refers to an identifier generated by the compiler to implement a feature such as iterator blocks or lambda expressions. The identifier is valid for the CLR, but not valid in C#: it's a name that cannot be "spoken" within the language, so it's guaranteed not to be clash with your code.

[9] We think, anyway. Eric can't remember for sure, and thinks Anders Hejlsberg *may* have come up with it first. I will always associate it with Eric though, along with his classification for exceptions: fatal, boneheaded, vexing or exogenous.

1.5 Summary

I love C#. It's both comfortable and exciting, and I love seeing where it's going next. I hope this chapter has passed on some of that excitement to you. But this has only been a taster. Let's get onto the real business of the book without further delay.