The C# Player's Guide

4th Edition Upgrade Companion

About This Document

This Upgrade Companion is meant for people who have the 4th Edition of *The C# Player's Guide* but want to use all the nice, new features in C# 10.

If you're still using C# 9 and .NET 5, you can just read the 4th Edition as it is.

If you have already read through the 4th Edition and are just wanting to see what features are new in C# 10, this document may be right for you, but you may also consider this article I have that describes these new features in depth:

https://csharpplayersguide.com/articles/whats-new-c-sharp-10

- Some items in this guide cover new things in C# 10, .NET 6, or Visual Studio 2022. Those are marked with [New].
- Some items are additional content about things that existed even in earlier versions of C#, but that deserved more attention. These are marked with [Additional Information].
- Some items represent mistakes in the book as printed. (Though I did not include every typo reported in this guide.) These are marked with [Correction].

Use this document as you read through the book. When you get to the noted paged number, stop and read the item for additional information about the topic.

Level 2: Getting an IDE

Page 12: Visual Studio Code Online [New]

There is a version of Visual Studio Code that runs online now: **vscode.dev**. Unfortunately, for now, it can't *run* code, just edit it. (Unless you buy CodeSpaces on GitHub.) Perhaps, someday, they'll be able to run the C# code directly in the browser.

Page 13: Visual Studio 2022 [New]

If you're using Visual Studio, the newest version is Visual Studio 2022. Many of the changes included in this document assume you're using at least Visual Studio 2022 (or the equivalent for the other IDEs) to get access to C# 10 and .NET 6 features. So you should download Visual Studio 2022 instead of Visual Studio 2019.

When installing, the workload you need is called **.NET desktop development**.

Page 14: You Don't Need to Sign In [More Information]

The book states that you will eventually need to make an account and sign in to Visual Studio. That is no longer needed (in Visual Studio 2019 or Visual Studio 2022).

Level 3: Hello World: Your First Program

Page 18: Use .NET 6 [New]

The book tells you to use .NET 5. For many features in the book, you'll want to use *at least* .NET 5. But to use the things in this Upgrade Companion, you'll want to use .NET 6.

Page 19: Hello, World! [New]

The template in C# 9 (and way back to the dawn of time) displayed `"Hello World!"` with no comma, while C# 10 templates add a comma: `"Hello, World!"`. This is more grammatically correct, but it has no impact on the code itself.

Page 20: Top-Level Statements [New]

We've arrived at our first substantial difference between what was and what now is. C# 9 introduced a new style of making an entry point or "main method" for your program. In a single file (not multiple across your whole program), you can add some bare statements. The 4th Edition tells you to delete all of the cruft that the template adds (`class Program` and `static void Main(string[] args)`) and replace it with a simple `System.Console.WriteLine("Hello World!");`.

You no longer need to delete that clutter. The new template doesn't add it in the first place.

The entire rest of 4th Edition assumes you're using this new style—called top-level statements—so there aren't a lot of other changes to the book due to this change.

Page 21: Implicit `using` Directives [New]

Everything described in the subsection called *`using` Directives* is entirely valid, but C# 10 has a significant change: several common `using` directives are automatically included in every file. That includes `System`, discussed here in Level 3. (In fact, nearly every namespace used in the book lands in this category.)

It is still essential to understand `using` directives and namespaces (though Level 33 covers both in more depth). You'll just be adding fewer of them to your programs.

So, you don't need to add `using System;` anymore, unless you're using older code or intentionally turn that feature off.

Level 6: The C# Type System

Page 44: Parse Methods [More Information]

After you finish the section about the `Convert` class, I want to also point out that many types have an alternative way to convert from a string to that type: a `Parse` method. For example:

```
int number = int.Parse("3");
```

This could be used as an alternative to:

```
int number = Convert.ToInt32("3");
```

The two essentially do the same thing, and you can use either style. In the book, I intentionally chose `Convert` because it is a single way to convert any of the built-in types to any of the other built-in types. It is less to remember. But if you prefer the `Parse` methods, feel free to use them instead.

Level 7: Math

Page 50: Comment Split Across Lines [Correction]

The comment at the start of the trapezoid code is inadvertently split across two lines. You'll get a compiler warning on the second line if you transcribe that code exactly as it appears in the book. (It unintentionally wrapped to the following line.) One way you could fix it would be to ensure the second line is also a comment:

```
// Some simple code for the area of a trapezoid:
// http://en.wikipedia.org/wiki/Trapezoid
```

Page 56: `Convert.ToInt` Is Wrong [Correction]

Toward the bottom of this page, there is a line that says, `int number = Convert.ToInt(text);`. But there is no `ToInt` method. This should be `ToInt32`.

Level 11: Looping

Page 81: "Forever Loops" [More Information]

The book states that an intentional infinite loop is sometimes called a *forever loop*. But other than some programming languages with a `forever` keyword, this is not a standard name for intentional infinite loops. I talked to many other programmers about this, and nobody could give me a name they'd use to refer to an intentional infinite loop. So I've removed it in future versions.

Page 84: Missing Quote [Correction]

In the first code sample under the section called *Nesting Loops*, there's a missing quotation mark in this line:

```
Console.WriteLine($"{a} * {b} = {a * b});
```

It should be:

```
Console.WriteLine($"{a} * {b} = {a * b}");
```

Level 13: Methods

Page 95: Local Functions [More Information]

The methods you make in Level 13 are all a particular type called a *local function*. Level 13 doesn't describe this in detail, but it is covered on page 259 in more depth.

But based on many conversations I've had with 4th Edition readers, I think it is important to have a decent understanding of local functions in Level 13.

Most methods live inside a type. `Console`'s `WriteLine` and `Convert`'s `ToInt32` methods are good examples of that. We aren't yet in the business of making types, but soon will be. Until we start making types, our methods need another home.

Methods can be defined inside of other methods. When they are, it is called a local function. That is what we do in Level 13, and the new methods we make are defined inside our main method as local functions. (The compiler places our main method into a `Program` class that we don't have to write out ourselves.)

At the bottom of page 95, there is a subsection called *Methods Get Their Own Variables*. This section describes how local functions can reach up to their container—the main method, in this case—and use its variables. I want to emphasize what the book says in this regard: avoid that for now. Don't let your local functions use the variables from their containing method. There's a time and place for that (see the section on closures on page 297), but it is not now. Use parameters and return values to share data between methods.

Page 99: Multiple Return Values? [More Information]

A question I have gotten several times is, how do I return multiple values? The short answer is, you can't. That may seem like a limitation, but we will soon see tools that let you get around this limitation.

Most notably, before long, we'll be making "composite types" that bundle two or more pieces of information into a single, cohesive bundle (tuples, classes, structs, records, etc.). Once we do this, we can return a composite type instead. This allows us to return a single thing that is made up of multiple parts.

On page 266, we will also see how to use output parameters, which is a second way to "return" more data.

So we're technically limited to a single return value but worry not; we can easily work around it as we collect more programming tools.

Level 17: Tuples

Page 137: Simula's Soups [Correction]

This challenge ended up being harder than I had intended. The main complexity is because the challenge expects you to make an array of tuples—your first exposure to tuples demands putting them in an array. The syntax for defining an array of tuples is covered in the book (see the `CreateHighScores` method on page 135), but it isn't very easy.

Feel free to modify this challenge to require making only a single soup, just as long as you store the actual soup and its constituent parts in a tuple (and not as separate variables). It will make your first experience with tuples go more smoothly. (And you still get credit for 100XP.)

Level 21: Static

Page 165: Factory Methods [More Information]

The book touches on factory methods, but I think it deserves to be fleshed out a bit more. Factory methods are mainly intended to be used outside of the class, while most of the samples here focus on using static things from within the class. The `CreateSquare` method defined on page 165 would be called like this from outside the class:

```
Rectangle r = Rectangle.CreateSquare(2);
```

This should look familiar. It is the same way we all `Console.WriteLine` and `Convert.ToInt32`, which are also defined as static methods.

Level 22: Null References

Page 169: Enlisting the Compiler's Help for Null Checking [New]

In C# 9, projects defaulted to not having the compiler help you check for null, and you had to turn it on (as shown on page 170) if you wanted it. In C# 10, that's on by default.

In C# 8 and earlier, as well as in C# 9, if you didn't reconfigure the project, writing out a reference type had no implications about nullability. For example:

```
string input = Console.ReadLine();
```

A type of `string` doesn't indicate whether `null` is an option or not. With C# 10 and the new project templates, a type of `string` means `null` is *not* an allowed option, while a type of `string?` (with the question mark) indicates `null` is considered a valid option.

With the new project templates (or with C# 9, if you turn it on), the compiler will give you warnings if you assign a null or something that might be null to something that expressly forbids it. Similarly, if you indicate something might be null, the compiler will give you warnings if you use something that might be null without a null check.

Perhaps the best, simple example of this is with `Console.ReadLine()`. `Console.ReadLine()`'s return type is `string?`, not `string`. You may have even noticed the compiler warning when we've done stuff like `string input = Console.ReadLine();`.

This change is one of the most significant ways the C# language has evolved in the last year. Most Internet code does not account for this. This feature didn't exist. So keep that in mind when you go to the Internet to look up how to do things.

Ultimately, this feature helps you catch many potential problems as long as you fix these compiler warnings.

In short: the 4th Edition describes how this feature works, but you no longer need to enable it. It is on by default. You should take advantage of it, using `?` to indicate `null` is allowed and leaving it off when it is not.

Level 28: Structs

Page 212: Field and Property Initializers and Constructors [New]

Near the bottom of page 212, there are two restrictions for structs that are no longer true in C# 10:

- "Because structs don't store references… there is always a parameterless constructor for a struct. Unlike a class, you cannot redefine it…."
- "Structs do not let you use field or property initializers…."

With C# 10, you can add field and property initializers and define a parameterless constructor. There's one crucial catch: because structs are value types, their memory exists without needing to be allocated independently. The existence of a value-typed variable is sufficient to ensure its memory exists, even without running a constructor. For example, assuming `Point` is a struct with public `X` and `Y` fields, you can do this:

```
Point p;
p.X = 3;
Console.WriteLine(p.X);
```

We never called a constructor for `p`, so any logic in your constructor won't run, including initializers.

Level 29: Records

Page 220: Records Can Now Be Classes or Structs [New]

With C# 10, you can make a record be either a class or a struct. The default is a class. But if you want it to be a struct, all you need to do is this:

```
public record struct Point(float X, float Y);
```

And you also have the option to specifically call out a record as being a class (though that isn't necessary) in a similar fashion:

```
public record class Point(float X, float Y);
```

If you make a struct-based record, the generated properties (like `X` and `Y` above) will have both getters and setters, unlike class-based records, which are `get` and `init`.

Level 31: The Fountain of Objects

Page 236: Wrong XP for the *Getting Help* Challenge [Correction]

This challenge should be 100 XP, not 50 XP. This change aligns it with the XP Tracker at the front of the book.

Level 32: Some Useful Types

Page 243: The `Count` Property [More Information]

Since it comes up later in the book, I want to call out that the `List<T>` type has a `Count` property that is philosophically the same thing as an array's `Length` property:

```
Console.WriteLine(words.Count);
```

Level 33: Managing Larger Programs

Page 255: File-Scoped Namespace Declarations [New]

After getting through the section on namespaces, you'll be interested to know about this C# 10 feature: file-scoped namespace declarations. Most files put all of their contents in a single namespace. If you're doing that, there's now a shorter way to express it. Instead of this:

```
namespace Something
{
    public class Type1 { ... }
    public class Type 2 { ... }
}
```

You can do this:

```
namespace Something;

public class Type1 { ... }
public class Type2 { ... }
```

This change gets rid of one extra layer of indentation, simplifying the code.

Page 258: Global `using` Directives [New]

After reading the section about `using` directives, I want to point out a related C# 10 feature: global `using` directives. You can put the `global` keyword on a `using` directive, and the compiler will treat it as though it were in every file in the project:

```
global using System.Text;
```

These must come before any other `using` directive in the file. However, I suggest not placing them in arbitrary files, or you'll never find them. Perhaps a *GlobalUsings.cs* file or something?

This feature makes sense if there's some namespace you're using everywhere in a project, but it can be easy to abuse. You don't want to slap a `global` on everything and call it a day. Every namespace you add clutters the accessible names. It gets hard to keep track of where everything is coming from, and you get more name collisions. I recommend only using a global `using` directive when you're using that namespace in the vast majority of the project's files.

I also want to point out that the book sometimes includes `using System;` in the samples. `System` seems like it would be a good candidate to add as a global `using` directive. But as I mentioned earlier in this document, several namespaces are automatically included in every .NET 6 project, so you won't generally need to write your own `global using System;`. It is already accounted for. The namespaces that are automatically included are:

- System
- System.Collections.Generic
- System.IO
- System.Linq
- System.Net.Http
- System.Threading
- System.Threading.Tasks

Level 34: Methods Revisited

Page 265: Swap Method is Wrong [Correction]

There is a minor error in the implementation of the `Swap` method. The variable, `b`, should be assigned the value in `temporary`, not `a`. The following is a better version of the method:

```
void Swap(ref int a, ref in b)
{
    int temporary = a;
    a = b;
    b = temporary;
}
```

Page 267: Challenge Should Use `TryParse(string s, out int result)` [Correction]

The Safer Number Crunching challenge says to use the `static int.TryParse(out int result)` method, but nothing with that specific signature exists. The intended one is `int.TryParse(string s, out int result)`, with a string parameter first.

Level 35: Error Handling and Exceptions

Page 273: Exception Handler, Not Event Handler [Correction]

Toward the bottom of the page, there's a paragraph that starts with, "When looking for an event handler…." This should read, "When looking for an *exception* handler…."

Level 37: Events

Page 289: Health <= 0 [Correction]

There are several times on pages 289 and 290 where the code says, `if (Health < 0)`. This code would be better if it were `Health <= 0` instead. Ships are destroyed when their health hits zero, not when it drops below zero.

Page 290: Dealing with Null Events [More Information]

With null reference checking on by default, most event types will likely indicate that they allow null as an option. Instead of using `Action`, you would generally use `Action?`, for example.

Page 290: Using Anonymous Methods to Avoid Null Values [More Information]

The book shows an example of assigning an event an empty method to ensure it never starts null with code like this:

```
public event Action ShipExploded = delegate { };
```

The book doesn't explain what that `delegate { }` thing does. That is a feature called an anonymous method, which is almost entirely replaced with lambdas (Level 38). Most people will allow an event to be null, but when they don't, this `delegate { }` thing still pops up from time to time.

If you want to learn a little more about anonymous methods, I wrote an article about it on the book's website: **https://csharpplayersguide.com/articles/anonymous-methods**. But few people use them, given the better syntax of lambdas. Lambda expressions are covered in Level 38, but if you want a lambda version of the above code, do the following:

```
public event Action ShipExploded = () => { };
```

Level 38: Lambda Expressions

Page 296: Explicit Return Types [New]

In addition to writing out parameter types when type inference fails, C# 10 lets you write out the return type:

```
int (int n) => n % 2 == 0
```

You shouldn't need to do this often (the compiler would have inferred an `int` return type from the above code anyway), but it has its uses.

Level 39: Files

Page 301: `WriteAllLines` Uses Any Collection, Not Just Arrays [More Information]

There is a sentence 2/3 of the way down this page that states, "[`File.WriteAllLines`] requires a `string[]` instead of a single `string`." There is an overload that uses `string[]`, but there is also one that uses `IEnumerable<string>`, meaning you could give it virtually any string collection (including `List<string>`), and it would work.

Page 304: Stream Examples [More Information]

I had a reader wonder about when you'd ever use a stream. The book touches on this but doesn't elaborate. So let me share a real-world situation where I needed streams instead of just using `File.WriteAllLines`.

Where I work, we have a logging database full of all sorts of important information. When we make an error report, we want to pull out large chunks of that data to include in the report. But it is a ton of data. Our initial solution was to query the database for everything we need, hang on to it in memory until we've got it all, and then save it to a file. This took up so much memory that the garbage collector would run 80% of the time, preventing critical systems from running fast enough. The fix: pull the data down in small chunks and write it to the file as we go. That memory can then get cleaned up without filling up too much memory. But you can't do that with `File.ReadAllLines`. You need a stream to write it to disk in small batches.

Streams are more complicated. Start with `File.WriteAllLines`/`WriteAllText` until it is clear you need something more flexible.

Level 40: Pattern Matching

Page 307: `when` not `where` [Correction]

The last item in the Speedrun says that case guards use the `where` keyword, but it is actually the `when` keyword. The main text on page 310 gets it right.

Page 311: Nested Pattern Improvements [New]

In the book, there's an example of a property pattern nested inside another property pattern:

```
Orc { Sword: { Type: SwordType.Longsword } }
```

This syntax is still valid (and nested patterns for a property are still a great tool in general), but if you're nesting one property pattern inside another, C# 10 gives you a shorthand for this:

```
Orc { Sword.Type: SwordType.Longsword }
```

Page 312: More about Positional Patterns [More Information]

The examples around positional patterns use a tuple which is broken apart into its parts:

```
(Choice.Rock, Choice.Scissors) => Player.One
```

In my experience, most positional patterns follow this form.

The book calls out (on page 313) that anything with a deconstructor can also use the positional pattern. But one thing that is left out is that a pattern can specify a type and variable name, in combination with the parentheses:

```
Dragon (DragonType.Black, LifePhase.Ancient) d => ...
```

If a type defines a deconstructor (and records usually do), you can use the positional pattern shown above to check the type and its deconstructed elements. The type and variable name are both optional.

Level 41: Operator Overloading

Page 316: Wrong Variable Name [Correction]

The code sample toward the bottom of this page doesn't compile. Instead of using the variable `a`, it should use `theLetterA`:

```
char theLetterA = 'a';
int theNumberA = (int)theLetterA;
```

Level 42: Query Expressions

Page 327: Ordering of HP is Backward [Correction]

The default order is ascending order, meaning the code on page 327 to calculate the strongest and weakest objects is backward. The first code sample in the Ordering section will calculate the weakest objects (the ones with the smallest `MaxHP`) first, and the second one will give you the strongest objects (with the largest `MaxHP`) first.

Level 47: Other Language Features

Page 368: The `System.Reflection` Namespace [More Information]

The book doesn't state this very clearly, but other than `Type`, which is in the `System` namespace, every other type discussed in this section is in the `System.Reflection` namespace. If you want to use reflection, you will almost certainly want to add a using directive for this namespace.

Page 372: Incorrect Bit Pattern [Correction]

We're looking at how the **<<** and **>>** operators work with a concrete example near the top of the page. The specifics of shifting right (taking the bit pattern `00010001` and shifting it three bits to the right) are written wrong. The book says, "`0001001` becomes `00000100`." But it should be, "`00010001` becomes `00000010`."

Level 50: .NET

Page 395: .NET 6 [New]

Perhaps this is obvious, but .NET 6 is now out. Use .NET 6 if you're following this Upgrade Companion.

Page 398: .NET Standard Fading Away [More Information]

The content about .NET Standard is still worth reading, but .NET Standard is not getting more updates in the future. .NET 5 and 6 are the beginning of a new era, generally superseding all versions of the .NET Framework and all versions of Mono. The driving force behind .NET Standard is mostly gone.

Level 55: Debugging Your Code

Page 441: Hot Reload [New]

The Edit and Continue feature described here is still a helpful feature. It allows you to edit a method you've paused execution in, make edits, and resume. Another feature called Hot Reload lets you edit any method, even without the program being paused, and apply the changes on the fly. The Hot Reload button is shown below and can be found on the toolbar, next to the green arrow button that starts your program.

Between the two, most code can be edited, and the changes applied to your code immediately.

There are two limitations. The first is that not all edits can be applied. Some won't work. But if you make an edit that can't be applied, it will tell you, and you can just close, recompile, and rerun the program. (And this list is getting shorter all the time.)

The second limitation is that Hot Reload seems only to take effect the next time a method is called. The current execution of a method will run to completion first. That's usually only a minor limitation. But it can be painful when you want to Hot Reload your main method. Leaving your main method means the program ends. So the Hot Reload feature won't help here, but hitting a breakpoint and pausing the program may allow you to make your edits. (And I wouldn't be surprised if even this limitation goes away in the future.)