THE C# PLAYER'S GUIDE

C# 12 EXPANSION

RB WHITAKER

The C# Player's Guide

C# 12 Expansion

About This Expansion

This C# 12 Expansion is meant for people with the 5th Edition of *The C# Player's Guide*. However, it should also work well for anybody with the 4th Edition or earlier.

This expansion covers the new C# 12 features. It spends more time on features impacting day-to-day programming and less on the corner case features.

This expansion contains two more challenges worth 200 total XP. Add these to your XP from the main book!

Read this expansion in parallel with the main book, or come back after you're done with the main book. Your choice. (The storyline in the challenge assumes you are doing it in parallel with the book rather than returning to it afterward.)

C# 12 is a minor update, especially compared to recent ones. The language design team worked on several large features but didn't finish them before the deadline. Still, the few features that made it are convenient improvements to C#.

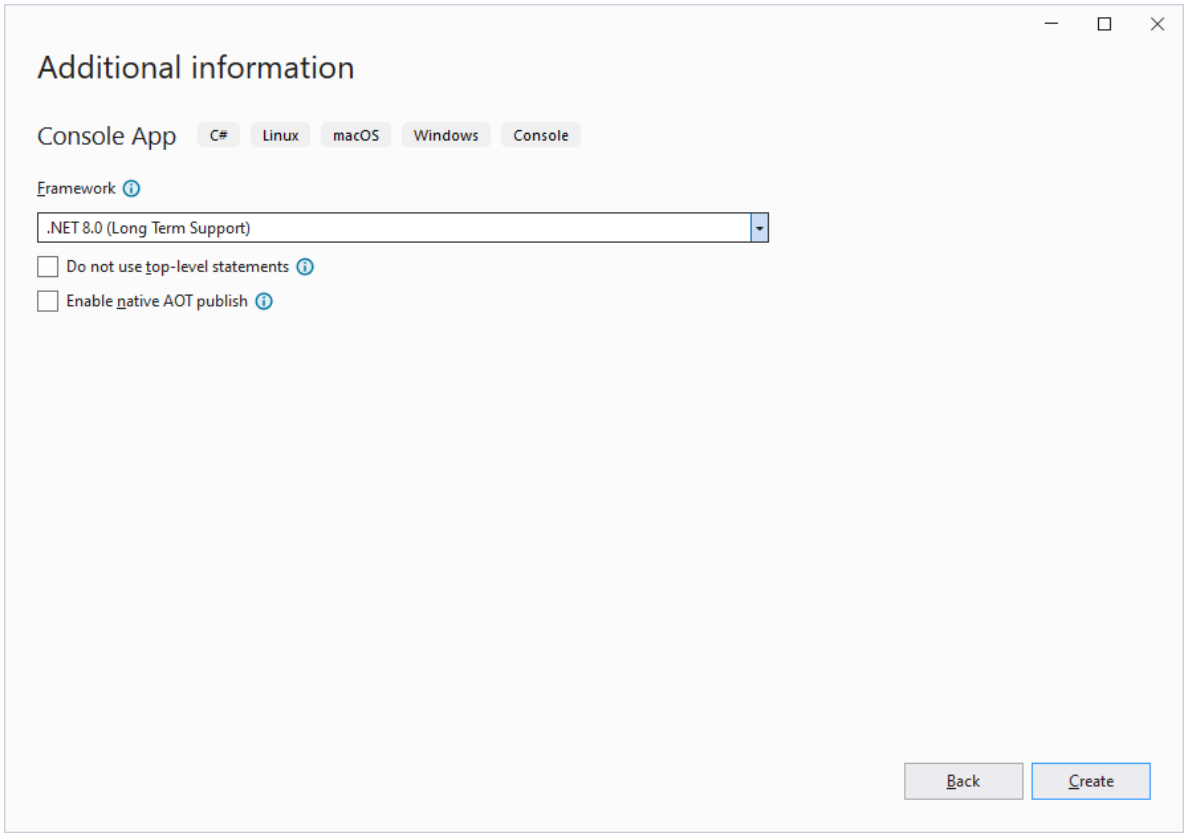
.NET 8 Projects

Read after Level 2: Getting an IDE.

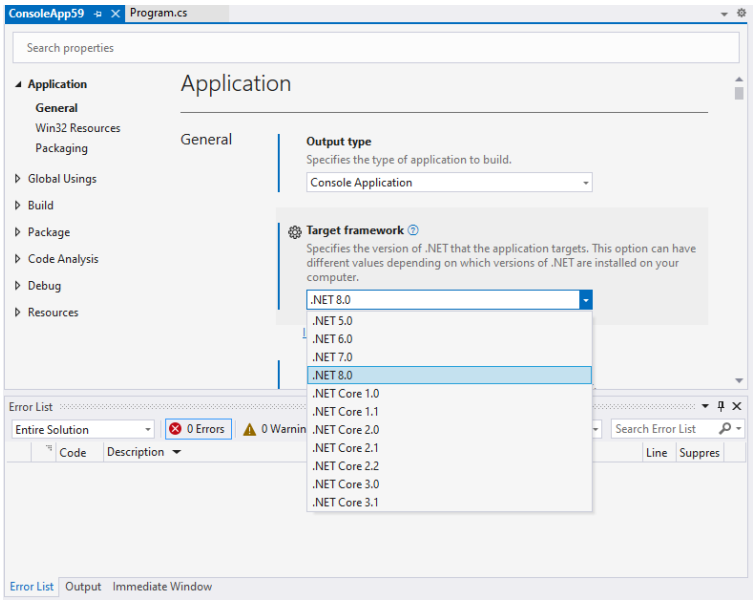
To use the features in this expansion, you must have .NET 8 installed.

For Visual Studio, if you already have the correct workflow installed (described in the book), you will need to update Visual Studio to 17.8 (or newer). If you have an older version, rerun the Visual Studio Installer and update it.

You will know that you have the correct version if you see .NET 8 (or newer) as an option when making a new project:



You can also upgrade an existing project to .NET 8 by right-clicking on the project in the Solution Explorer, choosing **Properties**, and then changing the **Target framework** setting:



Collection Expressions

Read after Level 12: Arrays.

C# gives you quite a few ways to define an array. The most formal is this:

```
int[] numbers1 = new int[5] { 2, 4, 6, 8, 10 };
```

This version requires stating the type, the amount, and the **`new`** keyword. In the early days of C#, this was the only way to create a new array, but over time, we picked up other options that allowed us to skip some of those elements if the compiler was able to infer them:

```
int[] numbers2 = new int[] { 2, 4, 6, 8, 10 };
int[] numbers3 = new [] { 2, 4, 6, 8, 10 };
```

C# 12 gives us yet another option that simplifies things even further, called a *collection expression*:

```
int[] numbers4 = [2, 4, 6, 8, 10];
```

I suspect this compact representation will quickly become commonplace because of its simplicity.

While we haven't yet discussed any collection type besides arrays, this syntax is not limited to arrays. We will eventually encounter other collection types that support it, including **`List<T>`** (Level 32) and anything that supports collection-initializer syntax.

This syntax also supports the *spread operator*, which uses the **`..`** symbol, just like the range operator. (The compiler can tell the difference depending on the context.) This operator lets you unpack an existing collection directly into another:

```
int[] group1 = [1, 2, 3];
int[] group2 = [2, 4, 6];
int[] group3 = [3, 6, 9];
int[] allNumbers = [..group1, ..group2, ..group3, 4, 8, 12];
```

The spread operator makes it trivial to make a new array as a copy of another, appending a few more values:

```
int[] numbers = [1, 2, 3];
numbers = [..numbers, 4];
```

These additional tools for making new arrays and other collection types make array-related code easier to read and write and are a fantastic addition to the language.

Challenge — Hunted — 100 XP

While traveling through Consolas late at night, a swarm of spiderlike machines, dispatched by the *Manticore*, spot you and begin to chase you. These hunter machines use light flashes to signal each other in the darkness. You get lucky and pin down one of the machines on a rooftop, away from the others, and inspect how it works. There appears to be a hibernation command in the machine that you think you could mimic, shutting down the whole swarm.

After a bit more analysis, it looks like the command consists of a flashing light with a brightness of 4, 8, 15, 16, 23, and then 42 lumens. This pattern must repeat forever to keep the swarm in hibernation.

Hurry up! You hear the spider machines skittering around the walls of the building below you. Time is running short!

Objectives:

- Create a program that makes an array containing the sequence of numbers 4, 8, 15, 16, 23, 42. You *must* use a collection expression, as shown above. Store this in a variable.
- Within a **`while (true)`** (or another infinite loop construct), display *only the first number* in the array (for example, **`Console.WriteLine(numbers[0]);`**).
- Within the loop, allocate a new array. Populate it with values from the current sequence of numbers, but move everything forward one slot while moving the current first item at the back of the array. The recently displayed number is now at the back of the array, and a new value is ready to be displayed.
- **Note:** You may use a simple **`for`** loop to populate the new array, but see if you can use the spread operator (discussed in the expansion above), a range operator (discussed in the book), and a collection expression.

Primary Constructors

Read after Level 29: Records.

How objects come into existence is a significant part of object-oriented programming. It should not come as a surprise that C# has many ways to define how objects are created, nor that more tools are added as the language evolves.

Traditionally, C# classes and structs were required to define their constructors as we've seen in the past, with a method-like member in the class definition like this:

```
public class Point
{
    public float X { get; }
    public float Y { get; }

    public Point(float x, float y)
    {
        X = x;
        Y = y;
    }
}
```

However, records got special treatment in the form of a constructor defined as a part of the class definition itself:

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

This **`Point`** type, defined as a record, will have a two-parameter constructor, even though we don't see it explicitly defined in the body of the type definition.

You may occasionally find such syntax useful, even if you are defining a normal, non-record class or struct, and with C# 12, this is now supported:

```
public class Point(float x, float y)
{
    public float X { get; } = x;
    public float Y { get; } = y;
}
```

With a primary constructor, field and property initializers can reference the parameters defined in the parentheses. However, unlike a record, such fields or properties will not be automatically created. You must declare them yourself. (If you want that, use a record.)

This example shows the key benefit of a primary constructor: the code just got much shorter. We don't need to write the constructor out; our initialization can be done inline.

On the other hand, if your constructor does anything beyond field and property initialization, you must resort back to a standard constructor to include that. Many constructors only do initialization and can benefit from a primary constructor, but not all of them.

There is also an element of stylistic preference. Some programmers prefer the traditional constructor for its versatility, uniformity, and familiarity. Others prefer primary constructors for their conciseness.

When using a primary constructor, note that all other constructors must chain back to the primary constructor using the **`this`** keyword.

```
public class Point(float x, float y)
{
    public float X { get; } = x;
    public float Y { get; } = y;
    public Point() : this(0, 0) { }
}
```

Challenge — A Night in the Wastelands — 100 XP

Your gift of Object Sight guides you toward the Fountain of Objects, taking you through the Wastelands, a frigid desert teeming with horrors of the Uncoded One's creation, though you've managed to steer clear until now. But as you set up camp for the night, a haunting howl pierces the silence. In a moment, you find yourself being chased over the rocks and dunes by a pack of coyote-like creatures, with rotting skin and muscle protected by a black patchwork carapace covering their bodies. You use your bow and loose a few code-infused arrows at the creatures, but the arrows seem slightly too slow to punch through the hard carapace. You scramble to the top of a stone hoodoo, safe for a moment but also trapped. The rotting coyotes are slowed but are still making their way up the rock toward you. It's just a matter of time before they get to you.

You look at your quiver of code-forged arrows and decide that if they were just a bit lighter, they'd be faster, which might be enough to pierce the carapace. The cracks in a few of the coyotes' carapace exteriors show that it shouldn't take much. The code within the arrows adds weight; if you can condense the code and shed the weight, you might survive.

Objectives:

- **Starting Point:** Either use your **`Arrow`** class from earlier in the book or start with the code below:

```
Arrow a = new Arrow(Arrowhead.Obsidian, Fletching.TurkeyFeathers, 78);
Console.WriteLine($"Arrowhead={a.Arrowhead} Fletching={a.Fletching} Length={a.Length}cm");

public class Arrow
{
    public Arrowhead Arrowhead { get; }
    public Fletching Fletching { get; }
    public float Length { get; }

    public Arrow(Arrowhead arrowhead, Fletching fletching, float length)
    {
        Arrowhead = arrowhead;
        Fletching = fletching;
        Length = length;
    }
}

public enum Arrowhead { Steel, Wood, Obsidian }
public enum Fletching { Plastic, TurkeyFeathers, GooseFeathers }
```

- Replace the constructor with a primary constructor without affecting how the rest of the program functions. Ensure all properties and fields are initialized through the primary constructor's parameters.
- **Answer this Question:** Generally, when writing software, the goal isn't to write it in as few lines as possible. Instead, the goal is to write code you can easily change, which requires writing code you can understand quickly. Sometimes, shorter code is easier to understand because there's less to read. Other times, shorter code is harder to understand because it is too compact or sheds helpful information. Compare your finished code against your starting point. The primary constructor made it shorter, but do you feel it made the code more or less readable?
- **After the Challenge:** After haphazardly remaking the code in your arrows, you send a few more down at the climbing coyotes. The plan worked. The arrows tear into the beasts, which tumble down the hoodoo's side to the sand below. After taking out a few more, the coyotes recognize they're in danger and flee, leaving you alone in the dark, still night.

Alias Enhancements

Read after Level 33: Managing Larger Programs.

In C# 12, the aliasing feature of using directives became more flexible. Before C# 12, you could do **`using Point = PhysicsEngine.Point;`**, and then in that file, **`Point`** refers to the one found in the **`PhysicsEngine`** namespace. However, the limitation was that the type on the right side of the equals sign had to be a formal, named type. Now, you can use it for virtually anything, including arrays and tuples:

```
using Point = (float X, float Y);
using Polygon = Point[];
```

Aliases can be a way to adjust and simplify names, but be cautious about overusing them. Such aliases aren't formalized and are simply masking the true name. Features like records make defining types for these things almost trivial while making them much more "real." When possible, I'll usually choose the following definitions over these aliases:

```
public record Point(float X, float Y);
public record Polygon(Point[] Points);
```

Read-Only Reference Parameters

Read after Level 34: Methods Revisited.

Before diving into this next feature, I want to clarify that this is advanced stuff you do *not* need to master yet. Even *basic* reference parameter usage is an advanced topic; just knowing how to call something like

`int.TryParse(out int result)` is all you'll need for a while. But C# 12 added more nuance to reference parameters, so I'll cover that here.

The **`ref`** and **`out`** keywords were the original tools for reference parameters. They share a variable's memory location instead of sharing the *contents* of the variable. Reference parameters are a powerful performance tool because they reduce the amount of copying in method calls, especially when used with large structs. They also give methods more power, as shown with the **`Swap(ref int a, ref int b)`** method in the book.

But sharing memory is dangerous. The method has full access to the memory location. C# requires you to put **`ref`** or **`out`** at the call site, confirming that you know you're sharing memory.

However, if a method only has a reference parameter for performance reasons—with no intention of *modifying* the memory—the danger is gone, and the overhead is overkill.

C# 7 added a third type of reference parameter. An input parameter, marked with **`in`**, is not allowed to modify the memory location. Input parameters are nice but have a limitation: changing a parameter from **`ref`** (C# 1.0) to **`in`** (added in C# 7) breaks code.

C# 12 solves this limitation by adding middle ground: read-only reference parameters. You can make a **`ref`** parameter a **`ref readonly`** parameter, which allows the method to claim it won't change the memory it has gained access to:

```
public void Display(ref readonly Point p) => Console.WriteLine($"({p.X}, {p.Y})");
```

For most practical purposes, this is identical to **`in`**. However, it doesn't break existing code because you call it like this:

```
Point somePoint = new Point(2, 3);
Display(ref somePoint);
```

Old, widely-used code can now be changed to **`ref readonly`** without breaking anything while guaranteeing it won't modify the memory.

You should now think of **`in`** as a variation on **`ref readonly`** with relaxed expectations. In particular, compare these two **`ref readonly`** calls, which both produce compiler warnings:

```
int x = 3;
Display(x); // No modifiers! Compiler warning (but not an error).
Display(5); // Passing in a value, not a variable! Compiler warning (but not an error).

void Display(ref readonly int number) => Console.WriteLine($"The number is {number}.");
```

The **`in`** equivalents do not produce any warnings:

```
int x = 3;
Display(x); // No modifiers! No warnings or errors.
Display(5); // Passing in a value, not a variable! No warnings or errors.

void Display(in int number) => Console.WriteLine($"The number is {number}.");
```

Said another way, **`in`** is like a **`ref readonly`** parameter that is more tolerant of skipping the modifiers and is happy to accept values in place of a variable. Those are not expected for a **`ref readonly`** parameter, but the compiler can make it work (thus producing a warning rather than an error).

Once again, reference parameters are complicated. You don't need to memorize all these details now, but being familiar with the options will make it easier to return to them when you eventually need them.

Optional Parameters in Lambdas

Read after Level 38: Lambda Expressions.

While only occasionally useful, methods defined with a lambda expression or statement now support optional parameters with a default value using the same syntax normal methods support. It requires a bit of a contrived example to illustrate, but the following shows its usage:

```
Action<int> function = (int x = 0) => Console.WriteLine(x);
function(10);
function();
```

The compiler is smart enough to spot that the lambda referenced by **`function`** has an optional parameter with a default value of 0, which supports calling it with or without that parameter. In the first case, with a specific argument of 10 passed in, that is the value used for **`x`** inside the method. In the second case, where no parameter is supplied, the compiler will call it with the default value of **`0`**, as you'd expect with an optional parameter.