THE C# PLAYER'S GUIDE

C# 11 EXPANSION

RB WHITAKER

The C# Player's Guide

C# 11 Expansion

About This Expansion

This C# 11 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# 11 features. It spends more time on features impacting day-to-day programming and less on the corner case features.

This expansion also includes details on topics I've seen readers struggle with. These sections don't represent new C# material but are helpful information for your journey.

This expansion contains four more challenges, worth a total of 450 XP. Add these to your total from the main book!

This expansion is meant to be read in parallel with the main book. Each section in this expansion begins by annotating when you should read the section, referencing the main book. The sections are ordered accordingly. Read the main book and jump over here when the time is right.

If you're already partway through the book, read the sections here until one references something beyond your current location. If you've completed the book, read through this expansion in any order.

.NET 7 Projects

Read after Level 2: Getting an IDE.

In the upcoming chapter, you'll see instructions for creating a project from the template. The book will mention ".NET 6 (or newer)." You're in that "or newer" part. Choose .NET 7 (or newer!) to use all of the features covered in this expansion.

You may also want to re-run the installer and ensure you have the latest version of everything.

Raw String Literals

Read after Level 8: Console 2.0.

While the most common way to write a string literal is simply enclosing the text in double quotes (**`"Hello"`**), it isn't so convenient when you have very long strings that span many lines. C# 11 introduces *raw string literals*, which are ideal for multi-line strings in your code:

```
string longText = """
    Pick an option:
    1. Up
    2. Down
    3. Left
    4. Right
    """;

Console.WriteLine(longText);
```

A raw string literal begins and ends with three quotation marks, which is how it differs from basic string literals.

Pay attention to the whitespace within that text, and observe what is displayed:

```
Pick an option:
1. Up
2. Down
3. Left
4. Right
```

You cannot put text on the first line of the raw string literal. Once the triple quotation marks appear, you must go to the following line immediately.

But notice that the leading whitespace on each line is skipped from the displayed string. This ensures you can control how indented the text is (so the code can be visually clean and organized) without including a bunch of extra whitespace in the string.

Technically, not all whitespace is ignored. The final line with the closing quotation marks governs the expected leading whitespace. It must always be on its own line, with no text before it. Every line before it must begin with identical whitespace, or it is a compiler error. The matching whitespace is eliminated, but any additional whitespace remains.

```
string longText = """
    Pick an option:
        1. Up
        2. Down
        3. Left
        4. Right
    """;
```

That will display the following:

```
Pick an option:
    1. Up
    2. Down
    3. Left
    4. Right
```

Raw string literals are "raw" because they don't allow escape sequences like **`\n`** for a new line, but there isn't a need, either. Rather than whitespace escape sequences like **`\n`** and **`\t`**, you can put a new line or tab directly in the text.

You also don't need to escape quotation marks that you want in the raw string literal. Since raw strings begin and end with three quotation marks, a single (or two) quotation marks can appear naturally in the string:

```
string longText = """
    "Thank you, Mario!" said Toad, "But our princess is in another castle!"
    """;
```

Using More Quotation Marks

So what about the rare case where you want three back-to-back quotation marks in a string?

Throw more quotation marks at the problem. A raw string literal must use *at least* three quotation marks to begin and end it, but you can use more. If you use five quotation marks, you can use up to four quotation marks within the string without terminating the string.

```
string longText = """""
    "
    ""
    """
    """"
    """""";
```

String Interpolation

Raw string literals support string interpolation as you'd expect:

```
int x = 2; int y = 3;
string longText = $"""
    X: {x}
    Y: {y}
    """;
```

But if you want to use string interpolation on a string that contains curly braces, you'll need to add more **$** symbols. For example, this code contains a snippet of the popular JSON data format, which naturally includes some strings:

```
string jsonText = """
    {
        "x": 2,
        "y": 3
    }
    """;
```

To keep those natural curly braces while also doing string interpolation, we'll use two (or more) **$** symbols. Then single curly braces are considered plain text, and two curly braces indicate string interpolation:

```
int x = 2; int y = 3;
string jsonText = $$"""
    {
        "x": {{x}},
        "y": {{y}}
    }
    """;
```

Add as many dollar signs as necessary if you need back-to-back curly braces to be plain text.

Challenge — Program-Making Program — 100 XP

You think back to the program you made for the triangle farmer, which computed the area of a triangle given its width and height. But you're unsettled about two things: (1) you may want to specify different units like meters, feet, or lightyears, and (2) you may want to use different types like **`int`**, **`float`**, and **`double`**. You have an idea: what if you made a program to write a triangle farmer program for you, where you can specify the units and the C# type you want to use and then embed those into a raw string literal to generate the code for you?

Objectives:

- **Note:** This program is very "meta." You will have actual C# code, and you will have a string that contains C# code inside it. It will look a little strange, but that's what comes with the territory for code generators.
- Create a program that asks for two inputs: the units to use (for example, "meters," "feet," or "lightyears") and a C# type (like "int," "float," and "decimal"). **Note:** You do not need to check that these are valid. Trust the user to pick a good unit and C# type.
- Use a multi-line raw string literal with interpolation to produce and display the following program, filling in the placeholders with the text the user entered in the previous bullet point.

```
Console.WriteLine("Enter the width (in [UNITS]) of the triangle: ");
[TYPE] width = [TYPE].Parse(Console.ReadLine());
```

```
Console.WriteLine("Enter the height (in [UNITS]) of the triangle: ");
[TYPE] height = [TYPE].Parse(Console.ReadLine());
[TYPE] result = width * height / 2;
Console.WriteLine($"{result} square [UNITS]");
```

- **Note:** You must modify the text above to make it work. The above is illustrative, not the actual text to put into a raw string literal.
- Run the program and enter good values for the units and type. Copy the output into a new program and run the generated program, now with your chosen units and C# type.
- **Note:** Your program does not need to run the final string. C# can execute arbitrary text as code, but it is tricky (and not secure). Take the text and copy/paste it into another program yourself.

Newlines in String Interpolation

Read after Level 8: Console 2.0.

On rare occasions, you may use a complex expression in string interpolation:

```
int x = 2, y = 3;
string text = $"They are {(x + 2) / (y - 1) * 5} when combined.";
```

Occasionally, these expressions are long enough to spread them over multiple lines. Before C# 11, that wasn't allowed. With C# 11, it is, and you can do stuff like this:

```
int x = 2, y = 3;
string text = $"They are {(x + 2) /
                          (y - 1) *
                          5} when combined.";
```

This change is a lovely refinement, removing an arbitrary limitation on what code could go inside a string interpolation expression.

But as a guideline, if my interpolated expressions are long enough to span multiple lines, I split the expression from the interpolated string anyway, like this:

```
int x = 2, y = 3;
int combined = (x + 2) / (y - 1) * 5;
string text = $"They are {combined} when combined.";
```

Enumerations and User Input

Read after Level 16: Enumerations but before the Simula's Test *challenge.*

This section doesn't cover new C# material, but it addresses a frequent question on the book's Discord server: how do you convert between strings and enumerations?

Let's initially assume the user types in good stuff. We'll deal with bad input later.

There are two strategies: handcrafted and generalized approaches. We'll look at conversions from strings to enumerations and enumerations to strings under each approach.

Handcrafted Conversions

The first option is to write code that converts specific enumeration values to strings or vice versa. In this approach, we must write code to consider the various possibilities, resulting in handcrafted code that works well for specific situations. Doing so is not hard and gives you complete control.

We start by deciding the mapping between enumeration values and strings.

Suppose we have this enumeration:

```
enum ElixirType { Invisibility, Strength, Regeneration, Coffee }
```

This enumeration represents an elixir type with four options.

We'll start by identifying how we want the user to see these, as shown in our user interface (the console window, for now). For example, we might represent **`ElixirType.Coffee`** as **`"coffee"`**, **`Strength`** as **`"strength"`**, **`Regeneration`** as **`"regeneration"`**, and **`Invisibility`** as **`"shadow walking"`**. I've intentionally picked something slightly strange on that last one to illustrate a point. The internal representation of something in code does not need to align with how you present it to the user.

The actual conversion code is typically straightforward. This is a great place for a switch expression:

```
string text = elixir switch
{
    ElixirType.Invisibility => "shadow walking",
    ElixirType.Strength     => "strength",
    ElixirType.Regeneration => "regeneration",
    ElixirType.Coffee       => "coffee"
};
```

However, I will often put this code in a method, as this task is well suited to a method:

```
string ConvertToString(ElixirType elixir) => elixir switch
{
    ElixirType.Invisibility => "shadow walking",
    ElixirType.Strength     => "strength",
    ElixirType.Regeneration => "regeneration",
    ElixirType.Coffee       => "coffee"
};
```

I used an expression body for this **`ConvertToString`** method, but a block body with a **`return`** also works.

Converting the other direction is as simple as flipping the switch around. However, you may want several strings to map to a single enumeration value. For example, **`"strength"`** and **`"Strength"`** could become **`ElixirType.Strength`**. You can add additional arms to your switch. Still, if it is a capitalization thing, you might also consider **`string`**'s **`ToLower()`** (or **`ToUpper()`**) method, and make the input lowercase, to disregard casing differences entirely:

```
string ConvertToElixir(string elixirText) => elixirText.ToLower() switch
{
    "shadow walking" => ElixirType.Invisibility,
    "invisibility"   => ElixirType.Invisibility,
    "strength"       => ElixirType.Strength,
    "regeneration"   => ElixirType.Regeneration,
    "coffee"         => ElixirType.Coffee
};
```

Alternatively, you could give the user a numbered menu and ask them to enter a number and use integers instead of strings.

For these examples, you'll note that the compiler gives you warnings for not covering all possible options. We'll deal with that in a moment.

Before moving on, let's address the downside of this approach. Because this is hand-coded, it may require maintenance over time. If you add or remove an enumeration value, you must revisit and update these switches to account for the changes.

Generalized Conversions

An alternative to handcrafted conversion code is to use a general-purpose conversion tool.

For example, you can call the **`ToString()`** method on any enumeration value:

```
ElixirType elixir = ElixirType.Invisibility;
Console.WriteLine(elixir.ToString());
```

The **`ToString`** method creates a string that exactly matches the name of the enumeration value in the code. So the code above will display **`"Invisibility"`**.

That "matches the code" thing is crucial. With a generalized solution, you have no direct control over the text. You lose the ability to display invisibility as "shadow walking." You cannot include spaces or apostrophes. You can't even control capitalization. If you wanted **`"invisibility"`**, you're out of luck.

This problem leads some people to hack the name of the code element. For example, they might make all enumeration values lowercase so that the produced text is **`"invisibility"`** and **`"strength"`**. I strongly discourage hacking the code to get the display you want for several reasons:

1. Enumeration values are UpperCamelCase by convention. Using a different scheme—especially when done inconsistently—makes the code harder to understand.
2. You may use enumeration values in different contexts and need different capitalization or punctuation in different places.
3. Not every string is a valid C# enumeration value identifier. If you want spaces or apostrophes, you're out of luck.

My advice is to either (a) live with the default capitalization and (lack of) punctuation that comes from this generalized approach or (b) use a handcrafted version instead.

In truth, the default capitalization is good enough for many low-stakes scenarios, including the challenges in this book. It's okay if your capitalization does not align perfectly with what the book shows in samples or the online solutions. But if you want things to match, don't hack the enumeration value names; go with the handcrafted version instead.

As a side note, I've seen people write code to make a generalized approach look nicer. For example, you might convert to a string with **`ToString()`** and add a space before every capital letter before making it all lowercase, making **`GoldenDragonsBreath`** become **`"golden dragons breath"`**. Feel free to do this if you want, but remember that it is tricky to cover all corner cases, so it is hard to perfect.

Many things automatically call **`ToString()`**, so you often don't need to. **`Console.WriteLine`** is an example. If you give **`Console.WriteLine`** the enumeration value you want to display, it will call **`ToString()`** on it, meaning you can do this:

```
ElixirType elixir = ElixirType.Invisibility;
Console.WriteLine(elixir);
```

Converting from a string to an enumeration is a bit trickier. There are generalized tools for this, but they use C# features we haven't covered yet. I will show you anyway because I think it is intuitive enough. But we'll leave the explanations until later.

In the past, we've seen how you can use **`int.Parse`** and similar methods to parse an **`int`** value. There is a similar method to parse an enumeration: **`Enum.Parse`**. The catch is that the **`Enum`** class doesn't know which specific enumeration type you want, so you need to state it. That is done using a feature called *generics*, which we'll spend a whole level discussing later. For now, I'll show the pattern:

```
string elixirText = Console.ReadLine();
ElixirType elixir = Enum.Parse<ElixirType>(elixirText);
```

The enumeration type is placed in angle brackets (**`<`** and **`>`**). Doing so gives the method enough information to know what type to return. Those angle brackets make **`Parse`** a *generic method*. Beyond showing an example, I won't bother explaining it further until later in the book. Swap out **`ElixirType`** for your desired enumeration type.

Note that the text must exactly match the enumeration value as it appears in the code. If your enumeration value is **`Invisibility`**, then **`"Invisibility"`** will work, but **`"invisibility"`** or **`"Invisible"`** won't.

Handling Bad Input

When converting between strings and enumerations, there's always some risk of bad values getting in there, especially if it involves a user.

The first strategy for dealing with bad input is to ignore it. If we ignore it and something happens, our program will crash. Some situations are low-stakes, and the effort to validate user input is not worth it. That's probably true for the challenges in the book, where you will likely be the only user and are smart enough to know how to re-run it and do it right the next time. Don't overlook this option. It is often the right tradeoff.

But if the stakes are high enough, you'll want to account for bad input. So let's discuss how you'd do so, starting with the handcrafted approach.

Our handcrafted approach used a switch to convert between enumeration values and strings. We can sometimes account for bad input by adding a discard pattern to catch anything not handled:

```
string text = elixir switch
{
    ElixirType.Invisibility => "shadow walking",
    ElixirType.Strength     => "strength",
    ElixirType.Regeneration => "regeneration",
    ElixirType.Coffee       => "coffee",
    _                       => "unknown"
};
```

This code ensures we don't crash by using some default value. We can go the other direction as well:

```
string ConvertToElixir(string elixirText) => elixirText.ToLower() switch
{
    "shadow walking" => ElixirType.Invisibility,
    "invisibility"   => ElixirType.Invisibility,
    "strength"       => ElixirType.Strength,
    "regeneration"   => ElixirType.Regeneration,
    "coffee"         => ElixirType.Coffee,
    _                => ElixirType.Coffee
};
```

What enumeration value should you pick in this situation? It depends. Sometimes, there's an obvious default. **`Coffee`** seems the least risky here, so it is a decent default. Sometimes, people will introduce an additional enumeration value called **`Default`**, **`Unknown`**, or something similar. The downside is that **`Unknown`** becomes a permanent, legitimate value, which is not always desirable.

Sometimes, you want to detect the invalid option and try again. Using **`Coffee`** doesn't give you enough information to know if the user intentionally chose **`Coffee`** or if it was a default. A specific **`Unknown`** or **`Default`** value does. We can put the switch in a loop and try until the chosen option is not **`Unknown`**.

Later in this expansion, we'll see how to use tuples (in the next level) to detect invalid inputs.

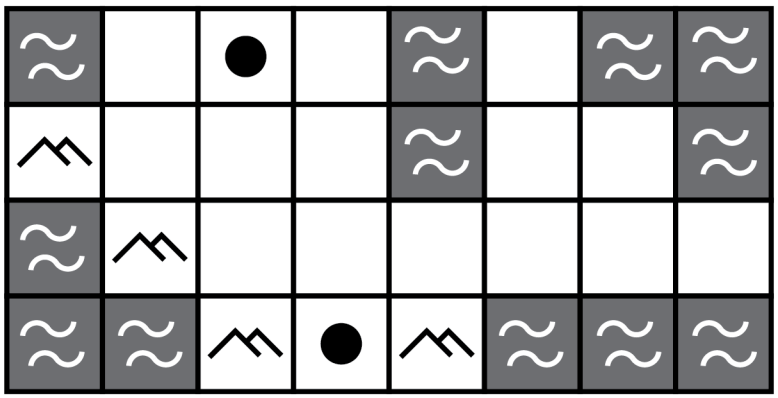
You don't have as much control over handling bad inputs for generalized solutions. The **`ToString()`** method will not crash because of bad input but may result in a number instead of useful text. There's nothing you can do about this.

The **`Enum.Parse`** method crashes if given invalid text, and you can't do anything about that. This mirrors how **`int.Parse`** or **`Convert.ToInt32`** works, so it shouldn't be surprising. In Part 3, we'll learn about the **`TryParse`** methods and how to use them to check for bad input.

Challenge — The Map — 150 XP

You've recently purchased a map of the island you've arrived at, but it isn't formatted well, and you know you can do better by putting the map data into a program. While you sit in a tavern, warmed by a large fire that is repelling a chill from the wind outside, sipping on a hot apple cider and munching on rustic bread and tangy cheese, you sit down to make a map program.

After carefully studying the map you've got, you sketch out on paper that this is what the map is supposed to look like when presented well:



The gray waves are water, and the others are different types of land. The empty spaces are plain land, the two inverted "v" symbols are mountains, and the dots are cities.

Objectives:

- Define an enumeration to represent the four types of terrain found in the map above.
- Create a 2D array that uses your new enumeration in your main method. Fill in its values to mirror the map shown above.
- Write a method to display the map above in the console window using **`"~~"`** to represent water, **`"^^"`** to represent mountains, **`"()"`** to represent cities, and **`"  "`** to represent open land.
- Set up your main method to call your new method, passing it the map array.
- **Hint:** Two nested **`for`** loops are a good way to loop through the whole map.
- **Hint:** Consider a switch expression to convert from the specific enumeration values to their text representations.

Tuples and Switches

Read after Level 17: Tuples.

Level 17 introduces tuples but also points out that once you learn some of the other upcoming tools (like classes, records, and structs), you won't use tuples all that often.

Tuples are a simple tool for small problems. If you want to represent an important concept throughout a program's lifetime, the other tools we'll learn about soon are usually the better choice. But tuples are great for temporary associations of two or more bits of data. Below, I'll share an example of a place where I might use tuples even after learning the other C# features from the coming pages.

The following code converts a string to an **`Elixir`** enumeration value. This code sidesteps concerns about handling an invalid string by creating a tuple containing a **`bool`**, representing successful conversion if **`true`**, and the converted enumeration value, which you should only look at if the conversion is successful.

```
ElixirType GetElixirType()
{
    while (true)
    {
        Console.WriteLine("Enter an elixir type.");
        string input = Console.ReadLine();
        (bool success, ElixirType elixir) = input switch
        {
            "shadow walking" => (true, ElixirType.Invisibility),
            "strength"       => (true, ElixirType.Strength),
            "regeneration"   => (true, ElixirType.Regeneration),
            "coffee"         => (true, ElixirType.Coffee),
            _                => (false, ElixirType.Invisibility)
        };

        if (success) return elixir;
        else Console.WriteLine("That was not a valid option. Try again.");
    }
}
```

The first four arms all use **`true`** in the first item of the tuple since the conversion worked. The last one is **`false`**. The value is irrelevant in the last arm because the conversion was unsuccessful. **`ElixirType.Invisibility`** is arbitrary.

While the switch produces a tuple, that tuple value is immediately unpacked into two distinct variables. That wasn't strictly necessary. The rest of the code could have accessed the items from the tuple. But this code felt simpler to me.

This temporary association of a **`bool`** and an **`ElixirType`** is a good place to use tuples. That remains true even after we learn about other tools in the upcoming levels.

Simula, Vin, and User Input

Read after Level 17: Tuples and before doing Simula's Soup.

The next two challenges, *Simula's Soup* and *Vin Fletcher's Arrows*, are tricky. I've answered many questions about these challenges in Discord. Without spoiling the fun, I want to provide extra content to put you on a decent footing when working on these challenges.

The key: **user input complicates these challenges significantly and is a secondary goal. Do it last.**

For *Simula's Soup*, start by making some tuples and displaying their contents. Perhaps start without naming the parts of your tuples (use `Item1`, `Item2`, etc.), skip the enumerations (use strings), and skip the user input. Get familiar with tuples first. Then replace the strings with enumerations. Then add the tuple element names. Finally, add the user input.

If you do, you'll end up with cleaner, simpler code and a better understanding of what tuples are for. (When you're all done, 75% of your code may be user input, but that's stuff we've been doing for a while now. So focus on tuples *without* user input first.)

For *Vin Fletcher's Arrows*, do the same thing. Skip the user input initially. Make an `Arrow` class with some fields, a constructor, and a method for computing the cost. Then make a few `Arrow` instances and display their costs and what they're made of (arrowhead type, fletching type, and length) so you can see how fields work. Make a few `Arrow` instances directly in code without user input by calling the constructor with specific values. Only after that works, add the user input.

I also recommend keeping the user input out of the `Arrow` class. Doing so will result in a more reusable `Arrow` class, making it easier to understand the value of objects and classes without complicating it with user input.

Required Members

Read after Level 20: Properties.

C# has two strategies for initializing an object: constructors and object-initializer syntax. The two can be used independently or together.

In a constructor-focused paradigm, if a field or property must be initialized, you add a constructor parameter for it:

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

This constructor guarantees that both `X` and `Y` are initialized.

The constructor-focused paradigm is my preferred mechanism for ensuring the initialization of fields and properties. I often use constructor parameters for anything mandatory and object-initializer syntax for optional things that have a reasonable default.

However, there are some limitations to this strategy. Plenty of C# programmers prefer object-initializer syntax. And in the future, we'll learn about inheritance (Level 25). In that paradigm, you can have classes that build upon other classes. If the one at the bottom suddenly needs a new property or field, you'll have to rework the entire pile of classes and their constructors to initialize it, which can be time-consuming.

C# 11 introduces the option to mark a property or field as `required`. When you do this, you state that the property *must* be assigned a value in object-initializer syntax. This is shown below:

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

With the **`required`** keyword added, the compiler demands that all new **`Point`** instances use object-initializer syntax to initialize **`X`** and **`Y`**:

```
Point p = new Point { X = 2, Y = 3 };
```

Required properties must also have an **`init`** (or **`set`**) accessor to be settable in object-initializer syntax.

Fields can also be marked **`required`**:

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

Public fields kill the benefits of information hiding and abstraction, so I try to avoid them (especially if they aren't **`readonly`**). But they do have occasional uses.

Required properties provide a tool to guarantee the initialization of properties without requiring going the constructor route.

AUTO-DEFAULT STRUCTS

Read after Level 28: Structs

Initializing structs got a little better in C# 11. Historically, any struct constructor has needed to initialize all fields within the struct. Failure to initialize the field resulted in a compiler error. With C# 11, the compiler will automatically add code to initialize any unassigned field to its default value. This mirrors classes.

```
public struct Point
{
    public float X;
    public float Y;

    public Point() { } // This now works in C# 11, initializing X and Y to 0.
}
```

It is worth reiterating that you can sidestep constructors when using structs, so this feature can still be bypassed.

```
Point p;
p.X = 3;
Console.WriteLine(p.Y); // COMPILER ERROR. `Y` is uninitialized.
```

STATIC INTERFACE MEMBERS

Read after Level 30: Generics.

C# 11 adds support for static members on an interface, meaning you can demand that all implementers provide a static method or property. (This also works for operators, which is discussed later.)

The mechanics are straightforward:

```
public interface ILevelGenerator
{
    Level CreateLevel(int number);
    static abstract int Count { get; }
}
```

The **`static`** keyword is natural since we're making a static member, but the **`abstract`** keyword may come as a surprise. Interface members are *naturally* abstract. Alas, due to conflicts with previous features, the **`abstract`** keyword is necessary to ensure they work as you'd expect.

But this feature looks more useful than it is. Interfaces are a tool primarily for creating interchangeable objects. When you use an interface type, you intentionally avoid committing to a specific type. To call a static member, you must go through the type name—**`RandomLevelGenerator.Count`**, for example.

There are only a few places where you can leverage static interface members. One such place is a generic method. For example, there is an **`IParseable<TSelf>`** interface that defines a static **`Parse`** method. Most built-in types like **`int`**, **`float`**, and **`byte`** implement this interface. You could make a generic method with a constraint limiting it to **`IParseable`**s. With this constraint, you can call this method through the generic type parameter:

```
public T[] ParseMany(string input) where T : IParsable<T>
{
    string[] parts = input.Split(',');
    T[] parsed = new T[parts.Length];

    for (int index = 0; index < parts.Length; index++)
        parsed[index] = T.Parse(parts[index]);

    return parsed;
}
```

This method can call the generic interface method, **`Parse`**, without committing to a specific type because it calls it through the generic type parameter **`T`**.

```
int[] a   = ParseMany<int>("1,2,3,4");
float[] b = ParseMany<float>("1,2.5,-9.81");
byte[] c  = ParseMany<byte>("0,255,127");
```

List Patterns

Read after Level 40: Pattern Matching.

C# 11 adds a few collection-centric pattern types known as *list patterns*. Suppose you monitor key presses and put each key in a **`List<ConsoleKey>`**:

```
List<ConsoleKey> keys = new ();
while (true)
{
    ConsoleKey key = Console.ReadKey(true).Key;
    if (key == ConsoleKey.Enter) break;
    keys.Add(key);
}
```

This code will loop and collect all key presses until you push the <**Enter**> key. We can now use pattern matching to respond to the input. The simplest of these is the empty list pattern, which is empty brackets (**`[]`**):

```
string response = keys switch
{
    [] => "Nothing",
    _  => ""
};
Console.WriteLine(response);
```

You can also match a specific single-item list like this:

```
string response = keys switch
{
    [] => "Nothing",
    [ConsoleKey.Spacebar] => "Space",
    _  => ""
};
```

The part between the brackets is a subpattern and can be anything. This happens to use the constant pattern.

We can match a whole set of items in the list by separating them with commas:

```
[ConsoleKey.W, ConsoleKey.A, ConsoleKey.S, ConsoleKey.D] => "WASD keys",
```

Order matters. This pattern will match W, A, S, then D, but not W, S, D, then A.

Lastly, you can use a **`..`** in your list pattern to indicate "zero or more other items here." For example:

```
[.., ConsoleKey.P] => "Ends with P",
[ConsoleKey.T, ..] => "Starts with a T",
[ConsoleKey.O, .., ConsoleKey.K, ConsoleKey.A, ConsoleKey.Y] => "Okay!",
```

The `..` can come at the beginning, middle, or end. Unfortunately, you can only have a single `..` in your list pattern, so there isn't a good way to express something like, "It must start with X, end with Y, and have a W somewhere in the middle."

While called *list* patterns, these patterns are not limited to only the **`List<T>`** type. If something has a **`Length`** or **`Count`** and can be indexed with **`[]`**, it will work, including arrays and other collection types.

Challenge — Premixed Potions — 100 XP

Complete after the Potion Masters of Pattren *challenge using your solution as a starting point.*

After creating a program to build programs one ingredient at a time, a couple of the younger apprentices seem to be avoiding eye contact with the masters. One of the masters notices their aversion and demands answers. "What trouble have you gotten yourselves into this time, Vel and Kitt?"

Vel, the taller of the two, replies sheepishly, "Well… it is just that we didn't realize we were supposed to make these slowly, one ingredient at a time. We just threw it all in there, all at once. Now, the Programmer's program won't help us, and we'll need to start over."

The master shakes her head before turning to you. "These two have a lot of learning to do, but, truthfully, even the masters sometimes get in a hurry and throw everything in at once. Could you make a version that identifies the right potion based on a list of ingredients? Like if I had water, stardust, and snake venom, it would just recognize that as a poison potion?"

Objectives:

- **Note:** Starting from your code for the *Potion Masters of Pattren* is a good starting point, but if you don't have it, you can make a new program from scratch (or use the solution online as a starting point).
- Carefully analyze the way potions are formed from the rules in the *Potion Masters of Pattren*, and come up with an ordered list of ingredients that each potion needs. For example, a poison potion is water, stardust, and snake venom. A flying potion is water, stardust, and dragon breath. Write these down on paper or in a code comment for later.
- Create a list or array to hold your potion ingredients, and add a few for a test.
- Use list patterns to detect which potion a list represents, including a catch-all for ruined potions, and display the final result.
- **Note:** Order matters in list patterns, which is mostly true of potions as well. However, cloudy potions can be made with different orders, and wraith potions build upon cloudy potions. You may need to add multiple patterns for the different ways to arrive at a cloudy or wraith potion (or get clever with **`when`** guards).
- **Note:** User input is *not* required for this challenge. You can make a list or array with specific hand-chosen items in code and test it by tweaking the version in code and recompiling and re-running it.
- **Note:** You do not need to make a single program that supports this new list-based model *and* the original item-by-item model. You can have two separate programs (or change what you had if you don't want to keep the first version around).

Generic Math

C# 11 added an interesting—albeit complex—bundle of features referred to as *generic math*. Generic math is not necessarily something you must master right away, but understanding the basics is worthwhile.

When writing math-related code, you will occasionally get stuck debating which of several number types to use. For example, you make a **`Point`** class and can't decide whether to use **`float`**s or **`double`**s. Or how about a **`Min`** method that returns the smallest of two values? You may want the minimum of two **`int`**s, **`double`**s, **`long`**s, or **`ushort`**s.

Traditionally, there have been two solutions:

1. Pick one and stick with it. I've worked in many codebases with a **`Point`** class. Generally, we settled on a single preferred option for the application or framework and lived with its problems to get its benefits.

2. Make multiple versions. If you cannot pick a single version, your alternative has been to make multiple versions. For an example, look no further than the **`Math.Min`** method group, which has overloads for each of the 11 built-in number types. Similarly, we could make **`PointF`**/**`PointD`** or **`FloatPoint`**/**`DoublePoint`** classes or structs.

One of these solutions is usually good enough. However, they are not without limits. If you need multiple versions, you'll maintain multiple nearly identical piles of code. And these are sometimes large. For example, MonoGame's **`Vector2`** struct spans 1200 lines of code. That repository chose to go with **`float`** and disregard **`double`**—a popular choice in game development circles. If they needed to maintain a **`double`**-based version, they'd have 2400 lines of nearly duplicate code to update, maintain, and debug. It is far from ideal.

C# has several tools for eliminating duplication in these situations. Let's discuss why they don't work here.

The first de-duplication tool is inheritance or interfaces. If there were a **`Number`** base class (or **`INumber`** interface) that **`int`**, **`double`**, etc., derived from, then you could arguably make a **`Point`** class or a **`Min`** method that used these instead of a specific type. There are a few problems with this:

1. You can't force the types to match. If **`Number`** is a base class of **`int`** and **`double`**, and your **`Point`** class uses **`Number`**, you may end up with an **`int`** for **`X`** and a **`double`** for **`Y`**.
2. The built-in number types are value types, so they can't use inheritance. Even if we used an **`INumber`** interface, storing value types in a reference-typed variable like **`INumber`** will box the value and place it on the heap, resulting in unnecessary memory allocations.
3. Until C# 11, interfaces could not include static members. As we've seen, operators are static. That means that even if we wanted to define an **`INumber`** interface, we wouldn't have been able to include the basic arithmetic operators on it that are commonly used with number types.

The second de-duplication tool is generics. Generics solve the first problem because you can make a **`Point<T>`** class that requires both **`X`** and **`Y`** to match. It also solves the boxing conversion problem because **`Point<int>`** and **`Point<double>`** use specific types, so they don't need to be moved to the heap in a boxing conversion. Alas, the third limitation applies to generics as well as inheritance.

But as we've seen, C# 11 supports static members to be included in an interface, including operator overloads. With C# 11, this third limitation is gone, and a new set of possibilities open up.

`INumber<T>` and Other `System.Numerics` Interfaces

First and foremost, now that C# allows you to define operators in an interface, C# 11 includes new interfaces that all the number types now implement. The simplest of these is **`INumberBase<T>`**, which requires that a type have the basic arithmetic operators (**`+`**, **`-`**, **`*`**, **`/`**, **`++`**, and **`--`**) and equality operators (**`==`** and **`!=`**). The **`INumber<T>`** interface augments that with the remainder (**`%`**) and comparison operators (**`>`**, **`<`**, **`>=`**, and **`<=`**).

These interfaces allow us to swap number types, solving this particularly tricky problem. For example, here is a generic **`Min`** method that works with any of the built-in number types:

```
public static T Min<T>(T a, T b) where T : INumber<T> => a < b ? a : b;
```

Note that you'll need to add a **`using`** directive to make this compile:

```
using System.Numerics;
```

`INumber<T>`, **`INumberBase<T>`**, and all other generic math interfaces are in the **`System.Numerics`** namespace, which is not automatically included.

This **`Min`** method has a generic type constraint requiring that **`T`** implement **`INumber<T>`**. This constraint ensures that the type—whatever it is—will have the **`<`** operator that this method uses. It can be used like this:

```
int smallest1  = Min<int>(2, 3);
double smallest2 = Min<double>(2.0, 3.0);
float smallest3  = Min<float>(2f, 3f);
```

Note that **`Min`** is a solved problem. The **`Math`** class provides overloads for all the different number types as an alternative solution. You won't need to write your own **`Min`** method. But this shows the mechanics on a small problem, which can be repeated on other problems.

While **INumber<T>** and **INumberBase<T>** are the most useful generic math interfaces, there are many more. There are generic interfaces for floating-point types, signed numbers, numbers with a known binary (2-based) representation, and more. I won't itemize the complete set here, but provide a link to explore more if you need these more advanced interface types: **https://learn.microsoft.com/en-us/dotnet/api/system.numerics**.

Operators in Interfaces

In C# 11, you can add static members to interfaces, which means you can add operators to an interface. You won't often need to do this yourself because of features we'll see in a moment. But it is worth walking through an example for illustration purposes. Here is a simple—but broken—take on an interface that requires an addition operator:

```
public interface IAdditionOperators // DOES NOT COMPILE
{
    static abstract int operator +(int a, int b);
}
```

While this looks good at a glance, there's a problem: operators can only be placed in a type involved in the operation. The addition defined above could only be added to the **int** type. We can fix that by using generics. The version below actually compiles:

```
public interface IAdditionOperators<T> where T : IAdditionOperators<T>
{
    static abstract T operator +(T a, T b);
}
```

The biggest surprise is the constraint, **where T : IAdditionOperators<T>**. That is noisy but necessary. That's the part that guarantees that the operator is only added to the **T** type, where it is allowed.

We could implement this interface like so:

```
public class Point : IAdditionOperators<Point>
{
    public T X { get; }
    public T Y { get; }
    public Point(T x, T y) { X = x; Y = y; }
    public static Point operator +(Point a, Point b) => new Point(a.X + b.X, a.Y + b.Y);
}
```

But this version of **IAdditionOperators** is missing a few things. It is common for an operator to take two values of the same type and produce a result of that same type. Add two **int**s to get another **int**. Add two **float**s to get another **float**. But those aren't hard limits. You can add two values of different types; the result does not need to match either. In other words, we don't have a single generic type parameter; we have *three*. So we'll evolve our interface definition for maximum flexibility:

```
public interface IAdditionOperators<TSelf, TOther, TResult>
                                        where TSelf : IAdditionOperators<TSelf, TOther, TResult>
{
    static abstract TResult operator +(TSelf a, TOther b);
}
```

This definition allows us to make an addition operator with mixed types if necessary, with the downside of needing to specify three generic type arguments that will usually be the same:

```
public class Point : IAdditionOperators<Point, Point, Point>
{
    // ...
}
```

This illustrates how you'd include an operator in an interface definition. It's complicated. The good news is, you won't need to do this often. In the **System.Numerics** namespace, right next to **INumber<T>** and **INumberBase<T>**, there is a set of per-operator interfaces much like our **IAdditionOperators** interface. Indeed, that example intentionally resembles the official **IAdditionOperators** definition.

Besides the official **IAdditionOperators** interface, there are other single-operator interfaces for each existing operator. **ISubtractionOperators** for **-**, **IMultiplicationOperators** for *****, **IIncrementOperators** for **++**, etc. A few operators are combined. **IEqualityOperators** is for **==** and **!=**, while **IComparisonOperators** is for **<**, **>**, **<=**, and **>=**. Once again, I won't dive into the details of every interface, but you can review the set here: **https://learn.microsoft.com/en-us/dotnet/api/system.numerics**

While **INumber<T>** and **INumberBase<T>** are more common, if you only need a single operator, you might consider one of these single-operator interfaces instead.

Challenge **Blast Damage** **100 XP**

The allies you've been gathering are beginning to assemble for the assault in the Uncoded One's domain. This has you wondering about the tools at your disposal, including Mylara and Skorin's magic cannon. You know it can damage anything near the impact site, but the farther away you get from it, the less damage will be dealt. Specifically, the splash damage will be **initialDamage / (distance * distance)**.

You want to give the gunners a program to help them quickly determine this damage. Alas, you don't want to commit to using a specific type like **float** or **int** because you may need to use different versions in different circumstances. But you also don't want to make multiple copies either.

Perhaps with generic math, you can solve this problem without duplication.

Objectives:

- Make a generic method with a type parameter of **T** that computes the damage, as indicated above. It should have a **T**-typed parameter for **initialDamage** and a **T**-typed parameter for **distance**, and it should also return a **T**.
- Add a type constraint to your generic method (see Level 30 for an example of placing a generic type constraint on a method) to ensure that whatever is used supports multiplication and division. You may use narrow, focused types like **IMultiplicationOperators** or a broader aggregate interface like **INumberBase**.
- **Note:** Don't forget to add a **using** directive for **System.Numerics**!
- In your main method, call your new method once using each of the following types: **double**, **float**, **decimal**, and **int**. For example, **Console.WriteLine(ComputeDamage(20f, 18f));**. (Parsing user input is not required.)

CHECKED AND UNCHECKED OPERATORS

Read after the Level 47 (Other Language Features) *section called* Checked and Unchecked Contexts.

Early in the book, we discussed overflow. If you add two large numbers, you can overflow the limits of your data type. For example:

```
int number = int.MaxValue;
number++;
```

What happens here? It depends. By default, the bits wrap around, and **number** becomes **int.MinValue**. What should have been a large positive number, mathematically, becomes a large negative number due to the limits of the computer.

In Level 47, we learned how to create a *checked context* and get these integer types to throw an exception instead of overflowing. It is almost as if there are two versions of the **+** operator—one for a checked context and one for an unchecked context.

You can define a checked-specific operator overload for your types if you have a compelling need to mimic this by defining a second operator overload and including the **checked** keyword in its definition:

```
public static Point operator checked +(Point a, Point b) { ... }
```

This is rarely needed. A single operator overload is usually good enough for checked and unchecked contexts. But with C# 11, you have it as an option if needed.

The File Access Modifier

Read after Level 47: Other Language Features

C# 11 introduces a new accessibility level: **`file`**. A type or member marked with **`file`** is accessible only within the file it is defined in.

```
file class SecretClass
{
    // ...
}
```

My recommendation is to use this access modifier sparingly.

The main purpose of **`file`** is code generators. When a code generator creates a type definition, the generator must pick a name. The generator does not want to conflict with anything you might use, which is a potential problem if this generated type is more broadly accessible. The file accessibility modifier ensures that code generators can pick a name that won't leak out or create conflicts.

Unless you're writing a code generator, use **`file`** sparingly. As a general principle, moving code around from one file to another should not break the code, and **`file`** has this potential problem.

Other Features

Read after Level 47: Other Language Features

The following is a summary of the other C# 11 features. I've kept their descriptions short because they're minor enhancements and corner cases you won't bump into often.

Generics and Attributes

Before C# 11, you could not make an attribute class generic. Now they can be:

```
public class PreferredValueAttribute<T> : Attribute
{
    public T PreferredValue { get; }
    public PreferredValueAttribute(T preferredValue)
    {
        PreferredValue = preferredValue;
    }
}
```

`IntPtr` and `UIntPtr` vs. `nint` and `nuint`

A few versions ago, C# introduced "native-sized" signed and unsigned integers: **`nint`** and **`nuint`**. These are keywords and work like **`int`** and **`uint`**. These integer types are 4 bytes on a 32-bit computer and 8 bytes on a 64-bit computer.

But C# has long had **`IntPtr`** ("int pointer" or "an integer the size of a pointer") and **`UintPtr`** types that were essentially the same thing, just with longer, more complex type names. With C# 11, **`nint`** and **`nuint`** are now aliases for **`IntPtr`** and **`UintPtr`**, similar to how **`int`** is an alias for **`Int32`**.

Extended `nameof` Scope

The **`nameof`** keyword lets you reference a code element to get its name as a string within your code. C# 11 makes **`nameof`** work within an attribute that you're applying to a method so that you can access the method name and the names of parameters for that method.

UTF-8 String Literals

Character encodings dictate how characters are stored in memory. A character encoding answers questions like the following: How many bytes do you use? What bits patterns represent which characters? Do all characters use the same number of bytes, or do some use more than others?

There are many character encodings used in computing. In C#, strings use the UTF-16 encoding. UTF-16 uses two bytes per character, though some "characters" occupy two **`char`** values. (Some emoji characters and Asian character sets will use two **`char`** values—four bytes—for conceptually single characters.) The main advantage of UTF-16 is that you can represent a vast set of characters (unlike old-school 1-byte character encodings like ASCII). Yet you can generally treat them as a fixed size, which makes a lot of text- and character-based code easy to implement.

The downside to UTF-16 is that most text primarily draws upon a limited set of characters—a-z, A-Z, 0-9, and some basic punctuation like brackets, commas, periods, etc. Those all fit in a single byte. Using two bytes for those all the time is wasteful. This limitation is especially notable when sending data across the Internet. The UTF-8 encoding is something of an Internet standard. It uses one byte for the most common characters, two for less common characters, and four for rare characters. The fact that you have to analyze each character to know how many characters there are (you can't just assume from the size of bytes) is its limitation, but it is a much more compact representation, which is great for data transfer.

The Base Class Library can convert the text using one encoding to another. But this UTF-16 vs. UTF-8 thing is very common. In C# 11, the language designers decided to make a shorthand way to define a string that the compiler automatically converts to UTF-8 rather than the standard UTF-16 that **`char`** and **`string`** use.

It is done by appending a **`u8`** to the end of a string:

```
ReadOnlySpan<byte> bytes = "Content-Type: text/html charset=utf-8"u8;
```

Note that this does *not* produce a **`string`** or **`char[]`**. It is a **`ReadOnlySpan<byte>`**.

This makes sense when you need small bits of hardcoded text to send across the Internet using UTF-8. You probably won't leverage this feature much if you're not doing that. Even with UTF-8 strings, you'll still use **`string`** and **`char`**. It doesn't eliminate the need.

Side note: **`Span<T>`** and **`ReadOnlySpan<T>`** are powerful optimization tools. However, they're also rather complex. For new programmers, my advice is to not worry too much about them yet, and study them later.

Pattern match `Span<char>` and `ReadOnlySpan<char>` with Strings

With C# 11, you can use **`Span<char>`** and **`ReadOnlySpan<char>`** with string literals:

```
Span<char> span = GetTextSpanFromSomewhere();
ElixirType type = span switch
{
    "Invisibility" => ElixirType.Invisibility,
    "Strength"     => ElixirType.Strength,
    "Regeneration" => ElixirType.Regeneration,
    _              => ElixirType.Coffee
};
```

This code works even though the constants are **`string`**s, and **`span`**'s type is **`Span<char>`**. This feature creates simple pattern-matching code without converting the **`Span<char>`** to a string.