LEVEL 33

MANAGING LARGER PROGRAMS

Speedrun

- C# programs can be spread across multiple files. It is common to put each type in its own file.
- Namespaces are a way to organize type definitions into named groups. Types intended for reuse should be placed in a namespace.
- You must normally refer to types by their fully qualified name, such as `System.Console`.
- A `using` directive allows you to use the simple name for a type instead of its fully qualified name.
- Several namespaces, including `System`, are automatically included in .NET 6+ projects and need no `using` directive. You can add to this list with a `global using` directive.
- A `using static` directive allows you to use static members of a type without the type name.
- Add types to a namespace with either `namespace Name { ... }` or by putting `namespace Name;` at the start of a file.
- Traditional entry points (before .NET 5) declare a `public static void Main(string[] args)` method in a `Program` class.

We've reached a critical point in our progress, and it is time to learn a few tools that will allow us to build even larger programs. We'll cover three topics here: splitting code across multiple files, namespaces, and traditional entry points.

USING MULTIPLE FILES

As your programs grow, having everything in a single file becomes unwieldy. You can spread C# code across many files and folders to organize your code. In fact, most C# programmers prefer putting types into separate files with a name that matches the type name. However, tiny type definitions like enumerations and records often get lumped in with closely related things.

There are many ways to get more files in your project (including **File** > **New** > **File...** or **Ctrl** + **N**). But sometimes, the easiest way is to initially put it into an existing file and then use a Quick Action in Visual Studio to move it to another file. The Quick Action will be available on any type you have defined in a file with a mismatched name. Imagine you have this code in *Program.cs*:

```
public class One { }
public class Two { }
```

You can get to the Quick Action by placing the cursor on the first line of a type definition (such as `public class One`), then clicking on the screwdriver or lightbulb icon or pressing **Alt + Enter**. When you do this, you will see a Quick Action named something like **Move type to One.cs**. Choosing this will create a new file (*One.cs*) and move the type there.

If a type is more than a few hundred lines long, it probably deserves its own file. Many C# programmers put each type in separate files; you're in good company if you do the same.

You can make as many files as you want with one caveat: a program can only contain one file with a main method. Every other file can only contain type definitions. If you have `Console.WriteLine("Hello, World!");` at the top of two files, the compiler won't know which to use as the entry point of your program.

NAMESPACES AND `using` DIRECTIVES

In C#, we name every type we create. Every other C# programmer is doing a similar thing. As you can imagine, some names are used more than once. For example, there are probably a hundred thousand different `Point` classes in the world.

To better differentiate identically named types from each other, we can place our types in a *namespace*. A namespace is a named container for organizing types. We can refer to a type by its *fully qualified name*, a combination of the namespace it lives in, and the type name itself. For example, since `Console` lives in the `System` namespace, its fully qualified name is `System.Console`.

Fully qualified names allow us to differentiate between types with the same simple name. We do similar things with people's names, especially when the name could be ambiguous or unclear: "Paul Leipzig," "Paul from work," or "Tall Paul."

Until now, we have not placed our types in a specific namespace. They end up in an unnamed namespace called the *global namespace*.

But most of the types we have used, such as `Console`, `Math`, and `List<T>`, all live specific namespaces. So far, everything we have covered has been in one of three namespaces.

The `System` namespace contains the most foundational and common types, including `Console`, `Convert`, all the built-in types (`int`, `bool`, `double`, `string`, `object`, etc.), `Math`/`MathF`, `Random`, `Random`, `DateTime`, `TimeSpan`, `Guid`, `Nullable<T>`, and tuples (`ValueTuple`). It is hard to imagine a C# program that doesn't use `System`.

The `System.Collections.Generic` namespace contains the generic collection types we discussed in Level 32, including `List<T>`, `IEnumerable<T>`, and `Dictionary<TKey, TValue>`. It is also hard to imagine any program with collections not using this namespace.

The `System.Text` namespace contains advanced text-related types, including the `StringBuilder` class we saw in Level 32. This namespace is not quite as common.

As you program in C#, you will encounter types defined in many other namespaces. These three are just the first three we've seen.

Any time you use a type name in your code, you have the option of using the type's fully qualified name. Since **`Console`**'s fully qualified name is **`System.Console`**, we could have written *Hello World* like this:

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

In fact, by default, you're *required* to use a type's fully qualified name! We saw an example of this in Level 32 when we talked about **`StringBuilder`**. The code there included this line:

```
System.Text.StringBuilder stringBuilder = new System.Text.StringBuilder();
```

So why haven't we had to write **`System.Console`** everywhere?

It's complicated.

You can include a line at the top of any file that tells the compiler, "I'm going to be using this namespace a lot. I want to use simple type names for things in this namespace. When you see a plain type name, look in this namespace for it." This line is called a **`using`** directive. For example:

```
using System.Text;

StringBuilder stringBuilder = new StringBuilder();
stringBuilder.Append("Hello, ");
stringBuilder.Append("World!");
Console.WriteLine(stringBuilder.ToString());
```

With that **`using`** directive at the top of the file, we no longer need to use **`StringBuilder`**'s fully qualified name when referring to that type.

In the future, you will want to pay attention to what namespace types live in, so you can either use their fully qualified name or add a **`using`** directive for their namespace. If you attempt to use a type's simple name without the correct **`using`** directive, you will get a compiler error because the compiler won't know what the identifier refers to.

These **`using`** directives partially explain why we don't always need to write out **`System.Console`**, but we haven't added **`using System;`** to our programs either. Why?

When you look at older C# code, you will find that they almost invariably start with **`using System;`** and a small pile of other **`using`** directives.

Starting with C# 10 projects, several **`using`** directives are added implicitly—you don't need to add them yourself. The automatic list includes both **`System`** and **`System.Collections.Generic`**, which we have encountered. It also includes **`System.IO`**, **`System.Linq`**, **`System.Net.Http`**, **`System.Threading`**, and **`System.Threading.Tasks`**, most of which we'll cover before the end of this book.

Because these extremely common namespaces are added implicitly, the pile of **`using`** directives at the start of a file only lists the non-obvious namespaces used in the file.

You can turn this feature off, but I recommend leaving it on, as it eliminates cluttered, obvious **`using`** directives across your code, which is a big win. On the other hand, if you're stuck in an older codebase and can't use this feature, you'll have to add **`using`** directives for every namespace you want to use or use fully qualified names.

For namespaces not in the list above, like **`System.Text`**, you will still need to add a **`using`** directive.

Advanced `using` Directive Features

The basic **`using`** directive, shown above, is what you will do most of the time. But there are a few advanced tricks you can do that are worth mentioning.

Global `using` Directives

If most files in a project use a specific namespace, you'll have **`using SomeNamespace;`** everywhere. As an alternative, you can include the **`global`** keyword on a **`using`** directive, and it will automatically be included in all files in the project.

```
global using SomeNamespace;
```

A global **`using`** directive can be added to any file but must come before regular **`using`** directives. I recommend putting these in a place you can find them easily. For example, you could make a *GlobalUsings.cs* or *ProjectSettings.cs* file containing only your global **`using`** directives.

Static `using` Directives

You can add a **`using`** directive with the **`static`** modifier to name a single type (not a namespace) to gain access to any static members of the type without writing out the type name. For example, the **`Math`** and **`Console`** classes have many static members. We could add static **`using`** directives for them:

```
using static System.Math;
using static System.Console;
```

With these in place, the following code compiles:

```
double x = PI;          // PI from Math.
WriteLine(Sin(x));      // WriteLine from Console, Sin from Math.
ReadKey();              // ReadKey from Console.
```

This leads to shorter code, but it does add a burden on you and other programmers to figure out where these methods are coming from. I recommend using these sparingly. More often than not, the burden of figuring out and remembering where the methods came from outweighs the few characters you save, but all tools have their uses.

Name Conflicts and Aliases

Suppose you want to use two types that share the same name in a single file. For example, imagine you need to use a **`PhysicsEngine.Point`** and a **`UserInterface.Point`** class. Adding **`using`** directives for those two namespaces results in a name conflict. The compiler won't know which one **`Point`** refers to.

One solution is to use fully qualified names to sidestep the conflict.

```
PhysicsEngine.Point point = new PhysicsEngine.Point();
```

Alternatively, you can also use the **`using`** keyword to give an alias to a type:

```
using Point = PhysicsEngine.Point;
```

The above line is sufficient for the compiler to know when it sees the type **`Point`** in a file, you're referring to **`PhysicsEngine.Point`**, not **`UserInterface.Point`**, which resolves the conflict.

An alias does not need to match the original name of the type, meaning you can do this:

```
using PhysicsPoint = PhysicsEngine.Point;
using UIPoint = UserInterface.Point;

PhysicsPoint p1 = new PhysicsPoint();
UIPoint p2 = new UIPoint();
```

Aliasing a type to another name can get confusing; do so with caution.

Putting Types into Namespaces

Virtually all types you use but don't create yourself (**`Console`**, **`Math`**, **`List<T>`**, etc.) will be in one namespace or another. Anything meant to be shared and reused in other projects should be in a namespace. If you are building something that isn't being reused, namespaces are somewhat less important. Everything we've done so far is in that category, so it isn't a big deal that we haven't used namespaces before.

But putting things into namespaces isn't hard and is often worth doing, even if you don't expect the code to be used far and wide.

The most flexible way of putting types in a namespace is shown below, using the **`namespace`** keyword, a name, and a set of curly braces that hold the types you want in the namespace:

```
namespace SpaceGame
{
    public enum Color { Red, Green, Blue, Yellow }

    public class Point { /* ... */ }
}
```

With this code, **`Color`**'s fully qualified name is **`SpaceGame.Color`**, and **`Point`**'s is **`SpaceGame.Point`**.

A slightly more complete example might look like this:

```
using SpaceGame;

Color color = Color.Red;
Point point = new Point();

namespace SpaceGame
{
    public enum Color { Red, Green, Blue, Yellow }
    public class Point { /* ... */ }
}
```

Our main method at the top isn't in the **`SpaceGame`** namespace, so it relies on the **`using`** directive at the top to use **`Color`** and **`Point`** without fully qualified names.

Namespaces can contain other namespaces:

```
namespace SpaceGame
{
    namespace Drawing
    {
    }
}
```

But the more common way to nest namespaces is this:

```
namespace SpaceGame.Drawing
{
}
```

A namespace can span many files. Each file will simply add to the namespace's members.

Aside from the file containing your main method, most files lump all of its types into the same namespace. The following version is a shortcut to say, "Everything in this file is in the **`SpaceGame`** namespace," allowing it to ditch the excessive curly braces and indentation:

```
namespace SpaceGame;

public enum Color { Red, Green, Blue, Yellow }
public class Point { /* ... */ }
```

This version comes after any **`using`** directives but before any type definitions. Unfortunately, you cannot use this version in the file containing your main method.

Namespace Naming Conventions

Most C# programmers will make their namespace names align with their project and folder structure. If you name your project SpaceGame, you would also make your namespace be **`SpaceGame`**. If you make a folder within your SpaceGame project called Graphics, you would put things in that folder in the **`SpaceGame.Graphics`** namespace.

Since namespace names usually mirror project names, let's briefly talk about project naming conventions. Project names are typically given a short, memorable project name (for example, **`SpaceGame`**) or prefix the project name with a company name (**`RBTech.SpaceGame`**). Some large projects are made of multiple components, so you'll sometimes see a component name added to the end (**`SpaceGame.Client`**, or **`RBTech.SpaceGame.Graphics`**). The namespace used within these projects will typically match these project names in each case.

Traditional Entry Points

Back in Level 3, I mentioned that there are two different ways to define an entry point for your program. Placing statements in a file like *Program.cs* is the simplest of the two and is what we have been doing in this book. This style is called *top-level statements* and is the newer and easier of the two options.

The alternative, which I'll call a *traditional entry point*, is still worth knowing. You will inevitably encounter code that still uses it, and if you find yourself using older code, it may be your only option.

The traditional approach is to make a class (usually called **`Program`**) and give it a **`static void Main`** method with an (optional) string array parameter (usually called **`args`**):

```
using System;

namespace HelloWorld
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

In fact, the newer top-level statement style is compiled into nearly identical code. Suppose you write this code:

```
Faction faction = PickFaction();
Console.WriteLine($"You've chosen to join the {faction} Faction.");

Faction PickFaction()
{
    Console.WriteLine("What faction do you want to be?");
    string? choice = Console.ReadLine();
    return choice switch
    {
        "Federation" => Faction.Federation,
        "Klingon"    => Faction.Klingon,
        "Romulan"    => Faction.Romulan,
    };
}

public enum Faction { Federation, Klingon, Romulan }
```

The compiler turns this code into the following:

```
internal class Program
{
    static void <Main>$(string[] args)
    {
        Faction faction = PickFaction();
        Console.WriteLine($"You've chosen to join the {faction} faction.");

        Faction PickFaction()
        {
            Console.WriteLine("What faction do you want to be?");
            string? choice = Console.ReadLine();
            return choice switch
            {
                "Federation" => Faction.Federation,
                "Klingon" => Faction.Klingon,
                "Romulan" => Faction.Romulan,
            };
        }
    }
}

public enum Faction { Federation, Klingon, Romulan }
```

A few notable points: (1) Instead of **`Main`**, it is called **`<Main>$`**, which is an “unspeakable” name that your code can’t refer to by name. (2) Your statements are placed in this generated main method. (3) Your methods are also put into the main method as local functions. (4) Any type you define is placed outside the main method and the **`Program`** class.

Knowledge Check — Large Programs — 25 XP

Check your knowledge with the following questions:

1. **True/False.** A **`using`** directive makes it so that you do not need to use fully qualified names.

2. What namespace are each of the following types in? (a) **`Console`** (b) **`List<T>`**, (c) **`StringBuilder`**.
3. What keyword is used to declare a namespace?
4. **True/False.** You should never write your own **`Program`** class and **`Main`** method.

Answers: **(1)** True. **(2)** (a) **`System`** (b) **`System.Collections.Generic`**, (c) **`System.Text`**. **(3)** **`namespace`**. **(4)** False.

Challenge The Feud 75 XP

On the Island of Namespaces, two families of ranchers are caretakers of the Medallion of Namespaces. They are in a feud. They are the iFields and the McDroids. The iFields ranch sheep and pigs and the McDroids ranch pigs and cows. Since both have pigs, they keep having conflicts. The two families will give you the Medallion of Namespaces if you can resolve the dispute and help them track their animals.

Objectives:

- Create a **`Sheep`** class in the **`IField`** namespace (fully qualified name of **`IField.Sheep`**).
- Create a **`Pig`** class in the **`IField`** namespace (fully qualified name of **`IField.Pig`**).
- Create a **`Cow`** class in the **`McDroid`** namespace (fully qualified name of **`McDroid.Cow`**).
- Create a **`Pig`** class in the **`McDroid`** namespace (fully qualified name of **`McDroid.Pig`**).
- For your main method, add **`using`** directives for both **`IField`** and **`McDroid`** namespaces. Make new instances of all four classes. There are no conflicts with **`Sheep`** and **`Cow`**, so make sure you can create new instances of those with **`new Sheep()`** and **`new Cow()`**. Resolve the conflicting **`Pig`** classes with either an alias or fully qualified names.

Challenge Dueling Traditions 100 XP

The inhabitants of Programain, guardians of the Medallion of Organization, seem to be hiding from you, peering at you through shuttered windows, leaving you alone on the dusty streets. The only other people on the road stand in front of you—a gray-haired wrinkle-faced woman and two toughs who stand just behind her. “We heard a Programmer might be headed our way. But you’re no True Programmer. In the Age Before, programmers declared their **`Main`** methods, used namespaces, and split their programs into multiple files. You probably don’t even know what those things are. Bah.” She spits on the ground and demands you leave, but you know you can win her and the townspeople over—and acquire the Medallion of Organization—if you can show you know how to use the tools she named. Do the following with one of the larger programs you have created in another challenge.

Objectives:

- Give your program a traditional **`Program`** and **`Main`** method instead of top-level statements.
- Place every type in a namespace.
- Place each type in its own file. (Small types like enumerations or records can be an exception.)
- **Answer this question:** Having used both top-level statements and a traditional entry point, which do you prefer and why?

LEVEL 34

METHODS REVISITED

Speedrun

- A parameter can be given a default value, which then makes it optional when called: `void DoStuff(int x = 4)` can be called as `DoStuff(2)` or `DoStuff()`, which uses the default of `4`.
- You can name parameters when calling a method: `DoStuff(x: 2)`. This allows parameters to be supplied out of order.
- `params` lets you call a method with a variable number of arguments: `DoStuff(params string[] words)` can be called like `DoStuff("a")` or `DoStuff("a", "b", "c")`.
- Use `ref` or `out` to pass by reference, allowing a method to share a variable's memory with another and to prevent copying data: `void PassByReference(ref int x) { ... }` and then `PassByReference(ref a);`. Use `out` when the called method initializes the variable.
- By defining a `Deconstruct` method (for example, `void Deconstruct(out float x, out float y) { ... }`) you can unpack an object into multiple variables: `(float x, float y) = point;`
- Extensions methods let you define static methods that appear as methods for another type: `static string Extension(this string text) { ... }`

Level 13 introduced the basics of methods. But that was only scratching the surface. Let's dig into some advanced usages of methods that all C# developers should know.

OPTIONAL ARGUMENTS

Optional arguments let you define a default value for a parameter. If you are happy with the default, you don't need to supply an argument when you call the method. Let's say you wrote this method to simulate rolling dice:

```
private Random _random = new Random();

public int RollDie(int sides) => _random.Next(sides) + 1;
```

This code lets you roll dice with any number of sides: traditional 6-sided dice, 20-sided dice, or 107-sided dice. The flexibility is nice, but what if 99% of the time, you want 6-sided dice?

Your code would be peppered with **RollDie(6)** calls. That's not necessarily bad, but it does make you wonder if there is a better way.

You could define an overload with no parameters and then just call the one above with a **6**:

```
public int RollDie() => RollDie(6);
```

With optional arguments, you can identify a default value where the method is defined:

```
public int RollDie(int sides = 6) => _random.Next(sides) + 1;
```

Only one **RollDie** method has been defined, but it can be called in either of these ways:

```
RollDie();   // Uses the default value of 6.
RollDie(20); // Uses 20 instead of the default.
```

You can have multiple parameters with an optional value, and you can mix them with normal non-optional parameters, but the optional ones must come last.

Optional parameters should only be used when there is some obvious choice for the value or usually called with the same value. If no standard or obvious value exists, it is generally better to skip the default value.

Default values must be *compile-time constants*. You can use any literal value or an expression made of literal values, but other expressions are not allowed. For example, you cannot use **new List<int>()** as a default value. If you need that, use an overload.

Named Arguments

When a method has many parameters or several of the same type, it can sometimes be hard to remember which order they appear in. **Math**'s **Clamp** method is a good example because it has three parameters of the same type:

```
Math.Clamp(20, 50, 100);
```

Does this clamp the value 20 to the range 50 to100 or the value 100 to the range 20 to 50?

When in doubt, C# lets you write out parameter names for each argument you are passing in:

```
Math.Clamp(min: 50, max: 100, value: 20);
```

This provides instant clarity about which argument is which, but it also allows you to supply them out of order. **Math.Clamp** expects **value** to come first, but it is last here.

You do not have to name every argument when using this feature; you can do it selectively. But, once you start putting things out of order, you can't go back to unnamed arguments.

Variable Number of Parameters

Look at this method that averages two numbers:

```
public static double Average(int a, int b) => (a + b) / 2.0;
```

What if you wanted to average three numbers? We could do this:

```
public static double Average(int a, int b, int c) => (a + b + c) / 3.0;
```

What if you wanted 5? Or 10? You could add as many overloads as you want, but it grows unwieldy fast.

You could also make **`Average`** have an **`int[]`** parameter instead. But that results in uglier code when you are calling it: **`Average(new int[] { 2, 3 })`** vs. **`Average(2, 3)`**.

The **`params`** keyword gives you the best of both worlds:

```
public static double Average(params int[] numbers)
{
    double total = 0;

    foreach (int number in numbers)
        total += number;

    return total / numbers.Length;
}
```

With that **`params`** keyword, you can call it like this:

```
Average(2, 3);
Average(2, 5, 8);
Average(41, 49, 29, 2, -7, 18);
```

The compiler does the hard work of transforming these methods with seemingly different parameter counts into an array.

You can also call this version of **`Average`** with an array if that is more natural for the situation.

You can only have one **`params`** parameter in a given method, and it must come after all normal parameters.

Combinations

You can combine default arguments, named arguments, and variable arguments, though I recommend getting used to each on their own before combining them.

Default arguments and named arguments are frequently combined. Imagine a method with four parameters, where each has a reasonable default value:

```
public Ship MakeShip(int health = 50, int speed = 26,
                     int rateOfFire = 2, int size = 4) => ...
```

You get a "standard" ship by calling **`MakeShip()`** with no arguments, taking advantage of all the default values. Or you can specify a non-default value for a single, specific parameter with something like **`MakeShip(rateOfFire: 3)`**. You get the custom value for the parameter you name and default values for every other parameter.

Passing by Reference

As we saw in Levels 13 and 14, when calling a method, the values passed to the method are duplicated into the called method's parameters. When a value type is passed, the value is copied into the parameter. When a reference type is passed, the reference is copied into the parameter. This copying of variable contents is called *passing by value*. In contrast, *passing by reference* allows two methods to share a variable and its memory location directly. The memory address itself is handed over rather than passing copied data to a method. Passing by reference allows a method to directly affect the calling method's variables, which is powerful but risky.

The terminology here is unfortunate. The concept of value types vs. reference types and the concept of passing by value vs. passing by reference are separate. You can pass either type using either mode. But as we'll see, passing by reference has more benefits to value types than it does to reference types.

Passing by reference means you do not have to duplicate data when methods are called. If you are passing large structs around, or even small structs with great frequency, this can make your program run much faster.

To make a parameter pass by reference, put the **`ref`** keyword before it:

```
void DisplayNumber(ref int x) => Console.WriteLine(x);
```

You must also use the **`ref`** keyword when calling the method:

```
int y = 3;
DisplayNumber(ref y);
```

Here, **`DisplayNumber`**'s **`x`** parameter does not have its own storage. It is an *alias* for another variable. When **`DisplayNumber(ref y)`** is called, that other variable will be **`y`**.

The primary goal is to avoid the costs of copying large value types whenever we call a method. While it achieves that goal, it comes with a steep price: the called method has total access to the caller's variables and can change them.

```
void DisplayNextNumber(ref int x)
{
    x++;
    Console.WriteLine(x);
}
```

If the above code used a regular (non-**`ref`**) parameter, **`x`** would be a local variable with its own memory. **`x++`** would affect only the new memory location. With **`ref`**, the memory is supplied by the calling method, and **`x++`** will impact the calling method.

This is typically undesirable—a cost that must be paid to get the advertised speed boosts. This risk is why you must put **`ref`** where the method is declared and where the method is called. Both sides must agree to share variables. But sometimes, it is precisely what you want. For example, this method swaps the contents of two variables:

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

Due to their nature, passing by reference can only be done with a variable—something that has a memory location. You cannot just supply an expression. You cannot do this:

```
DisplayNumber(ref 3); // COMPILER ERROR!
```

Passing by reference is primarily for value types. Reference types already get most of the would-be benefits by their very nature. But reference types can also be passed by reference.

Methods assume parameters are initialized when the method is called. You must initialize any variable that you pass by reference before calling it. The code below is a compiler error because **`y`** is not initialized:

```
int y;
DisplayNumber(ref y); // COMPILER ERROR!
```

Output Parameters

Output parameters are a special flavor of **`ref`** parameters. They are also passed by reference, but they are not required to be initialized in advance, and the method must initialize an output parameter before returning. Output parameters are made with the **`out`** keyword:

```
void SetupNumber(bool useBigNumber, out double value)
{
    value = useBigNumber ? 1000000 : 1;
}
```

Which is called like this:

```
double x;
SetupNumber(true, out x);
double y;
SetupNumber(false, out y);
```

Mechanically, output parameters work the same as reference parameters. But as you can see, neither **`x`** nor **`y`** was initialized beforehand. This code expects **`SetupNumber`** to initialize those variables instead.

Output parameters are sometimes used to return more than one value from a method. You will find plenty of code that does this, but also consider returning a tuple or record since these sometimes create simpler code.

When invoking a method with an output parameter, you can also declare a variable right there, instead of needing to declare it on a previous line:

```
SetupNumber(true, out double x);
SetupNumber(false, out double y);
```

You will also encounter scenarios where the method you're calling has an output parameter that you don't care to use. Instead of a throwaway variable like **`junk1`** or **`unused2`**, you can use a discard to ignore it:

```
SetupNumber(true, out _);
```

One notable usage of output parameters appears when parsing text. As we saw in Level 6, most built-in types have a **`Parse`** method: **`int x = int.Parse("3");`**. If these methods are called with bad data, they crash the program. These types also have a **`TryParse`** method, whose return value tells you if it was able to parse the data and supplies the parsed number as an output parameter:

```
string? input = Console.ReadLine();
if (int.TryParse(input, out int value))
    Console.WriteLine($"You entered {value}.");
else
    Console.WriteLine("That is not a number!");
```

There's More!

SIDE QUEST

Passing by reference is a powerful concept. You will find the occasional use for it. But what we have covered here is only scratching the surface. The details are beyond this book, getting into the darkest corners of C#. But just so you have an idea of what else is out there in these deep, dark caverns, here are a few hints about how else passing by reference can be used.

Most of the time, the memory location shared when passing by reference is owned by the calling method. The called method can originate a shared memory location using *ref return values*. The rules are complex because they must ensure that the memory location returned to the calling method will still be around after returning. There are also *ref local variables* that function as local variables but are an alias for another variable.

You can also make a pass-by-reference input parameter with the **`in`** keyword. This keyword hints that the method will not modify the variable passed to the method, but how it ensures this is not straightforward. The compiler can easily enforce that you never assign a completely new value to the supplied variable. The rest is trickier. The compiler does not magically know which properties and methods will modify the object and which won't. To ensure the called method doesn't accidentally change the **`in`** parameter, it will duplicate the value into another variable and call methods and properties on the copy instead. But bypassing those duplications was a key reason for passing by reference in the first place, which somewhat defeats the purpose. To counter that, you can mark some structs and some struct methods as **`readonly`**, which tells the compiler it is safe to call the method without making a defensive copy first.

This sharing of memory locations is also the basis for a special type called **`Span<T>`**, representing a collection that reuses some or all of another's memory.

Challenge — Safer Number Crunching — 50 XP

"Master Programmer! We need your help! We are but humble number crunchers. We read numbers in, work with them for a bit, then display the results. But not everybody enters good numbers. Sometimes, we type in wrong things by accident. And sometimes, somebody does it *on purpose*. Trolls, looking to cause trouble, I tell ya!

"We've heard about these so-called **`TryParse`** methods that cannot fail or break. We know you're here looking for Medallions and allies. If you can help us with this, the Medallion of Reference Passing is yours, and we will join you at the final battle."

Objectives:

- Create a program that asks the user for an **`int`** value. Use the static **`int.TryParse(string s, out int result)`** method to parse the number. Loop until they enter a valid value.
- Extend the program to do the same for both **`double`** and **`bool`**.

Deconstructors

With tuples, we can unpack the elements into multiple variables simultaneously:

```
var tuple = (2, 3);
(int a, int b) = tuple;
```

You can give your types this ability by defining a **Deconstruct** method with a **void** return type and a collection of output parameters. The following could be added to any of the various **Point** types we have defined:

```
public void Deconstruct(out float x, out float y)
{
    x = X;
    y = Y;
}
```

While you can invoke the **Deconstruct** method directly (as though it were any other method), you can also call it with code like this:

```
(float x, float y) = p;
```

By adding **Deconstruct** methods, you give any type this deconstruction ability. This is especially useful for data-centric types. (Records have this automatically.)

You can define multiple **Deconstruct** overloads with different parameter lists.

Extension Methods

An *extension method* is a static method that can give the appearance of being attached to another type (class, enumeration, interface, etc.) as an instance method. Extension methods are useful when you want to add to a class that you do not own. They also let you add methods for things that can't or typically don't have them, such as interfaces or enumerations.

For example, the **string** class has the **ToUpper** and **ToLower** methods that produce uppercase and lowercase versions of the string. If we wanted a **ToAlternating** method that alternates between uppercase and lowercase with each letter, we would normally be out of luck. We don't own the **string** class, so we can't add this method to it. But an extension method allows us to define **ToAlternating** as a static method elsewhere and then use it as though it were a natural part of the **string** class:

```
public static class StringExtensions
{
    public static string ToAlternating(this string text)
    {
        string result = "";

        bool isCapital = true;
        foreach (char letter in text)
        {
            result += isCapital ? char.ToUpper(letter) : char.ToLower(letter);
            isCapital = !isCapital;
        }

        return result;
    }
}
```

As shown above, an extension method must be static and in a static class. But the magic that turns it into an extension method is the **this** keyword before the method's first parameter. You can only do this on the first parameter.

When you define an extension method like this, you can call it as though it were an instance method of the first parameter's type:

```
string message = "Hello, World!";
Console.WriteLine(message.ToAlternating());
```

It is typical (but not required) to place extension methods for any given type in a class with the name **`[Type]Extensions`**. We defined an extension method for the **`string`** class, so the class was **`StringExtensions`**.

Extension methods can have other parameters after the **`this`** parameter. They are treated as normal parameters when calling the method. So **`ToAlternating(this string text, bool startCapitalized)`** could be called with **`text.ToAlternating(false);`**.

Extension methods can only define new instance methods. You cannot use them to make extension properties or extension static methods.

Knowledge Check — Methods — 25 XP

Check your knowledge with the following questions:

Use this for questions 1-3: `void DoSomething(int x, int y = 3, int z = 4) { ... }`

1. Which parameters are optional?
2. What values do **x**, **y**, and **z** have if called with **`DoSomething(1, 2);`**
3. What values do **x**, **y**, and **z** have if called with the following: **`DoSomething(x: 2, z: 9);`**
4. **True/False**. You must define all optional parameters after all required parameters.
5. **True/False**. A parameter with the **`params`** keyword must be the last.
6. What keyword is added to a parameter to make an extension method?
7. What keyword indicates that a parameter is passed by reference?
8. Given the method **`void DoSomething(int x, params int[] numbers) { ... }`** which of the following are allowed? (a) **`DoSomething();`** (b) **`DoSomething(1);`** (c) **`DoSomething(1, 2, 3, 4, 5);`** (d) **`DoSomething(1, new int[] { 2, 3, 4, 5 });`**

Answers: **(1)** y and z. **(2)** x=1,y=2,z=4. **(3)** x=2,y=3,z=9. **(4)** True. **(5)** True. **(6)** **`this`**. **(7)** **`ref`** or **`out`** **(8)** b, c, d.

Challenge — Better Random — 100 XP

The villagers of Randetherin often use the **`Random`** class but struggle with its limited capabilities. They have asked for your help to make **`Random`** better. They offer you the Medallion of Powerful Methods in exchange. Their complaints are as follows:

- **`Random.NextDouble()`** only returns values between 0 and 1, and they often need to be able to produce random **`double`** values between 0 and another number, such as 0 to 10.
- They need to randomly choose from one of several **`string`**s, such as **`"up"`**, **`"down"`**, **`"left"`**, and **`"right"`**, with each having an equal probability of being chosen.
- They need to do a coin toss, randomly picking a **`bool`**, and usually want it to be a fair coin toss (50% heads and 50% tails) but occasionally want unequal probabilities. For example, a 75% chance of **`true`** and a 25% chance of **`false`**.

Objectives:

- Create a new static class to add extension methods for **`Random`**.

- As described above, add a **`NextDouble`** extension method that gives a maximum value for a randomly generated **`double`**.
- Add a **`NextString`** extension method for **`Random`** that allows you to pass in any number of **`string`** values (using **`params`**) and randomly pick one of them.
- Add a **`CoinFlip`** method that randomly picks a **`bool`** value. It should have an optional parameter that indicates the frequency of heads coming up, which should default to 0.5, or 50% of the time.
- **Answer this question:** In your opinion, would it be better to make a derived **`AdvancedRandom`** class that adds these methods or use extension methods and why?

LEVEL 35

ERROR HANDLING AND EXCEPTIONS

Speedrun

- Exceptions are C#'s primary error handling mechanism.
- Exceptions are objects of the `Exception` type (or a derived type).
- Put code that may throw exceptions in a `try` block, and place handlers in `catch` blocks: `try { Something(); } catch (SomeTypeOfException e) { HandleError(); }`
- Throw a new exception with the `throw` keyword: `throw new Exception();`
- A `finally` block identifies code that runs regardless of how the `try` block exits—exception, early return, or executing to completion: `try { ... } finally { ... }`
- This level contains several guidelines for throwing and catching exceptions.

We have been pretending nothing will ever go wrong in our programs, and it is time to face reality. What should we do when things go wrong? Consider this code that gets a number from the user between 1 and 10:

```
int GetNumberFromUser()
{
    int number = 0;

    while (number < 1 || number > 10)
    {
        Console.Write("Enter a number between 1 and 10: ");
        string? response = Console.ReadLine();
        number = Convert.ToInt32(response);
    }

    return number;
}
```

What happens if they enter "asdf"? `Convert.ToInt32` cannot convert this, and our program unravels. Under real-life circumstances, our program crashes and terminates. If you are running in Visual Studio with a debugger attached, the debugger is smart enough to recognize that a crash is imminent and pause the program for you to inspect its state in its death throes.

In C#, when a piece of code recognizes it has encountered a dead end and cannot continue, a kind of error called an *exception* can be generated by the code that detects it. Exceptions bubble up from a method to its caller and then to that method's caller, looking to see if anything knows how to resolve the problem so that the program can keep running. This process of transferring control farther up the call stack is called *throwing* the exception. Parts of your code that react to a thrown exception are *exception handlers*. Or you could say that the exception handler *catches* the exception to stop it from continuing further.

HANDLING EXCEPTIONS

Most of our code can account for all scenarios without the potential for failure—for example, **`Math.Sqrt`** can safely handle all square roots. (Though it does produce the value **`double.NaN`** for negative numbers.) This is the ideal situation to be in. Success is guaranteed.

On the other hand, **`Convert.ToInt32`** makes no such guarantee. When called with **`"asdf"`**, we encounter the problem. The text cannot be converted, and the method cannot proceed with its stated job. Our approach for dealing with such errors has previously boiled down to, "Dear user: Please don't do the dumb. I can't handle it when you do the dumb." Then cross your fingers, put on your lucky socks, and grab your *Minecraft* Luck of the Sea enchantment.

Rather than hoping, let's deal with this issue head-on. We must first recognize that a code section might fail and also have a plan to recover. The problem code is placed in a **`try`** block, immediately followed by a handler for the exception:

```
try
{
    number = Convert.ToInt32(response);
}
catch (Exception)
{
    number = -1;
    Console.WriteLine($"I do not understand '{response}'.");
}
```

The **`catch`** block will catch any exception that arises from within the **`try`** block, and the code contained there will run so that you can recover from the problem. In this case, if we fail to convert to an **`int`** for any reason, we will display the text **`"I do not understand..."`** and set **`number`** to **`-1`**.

Let's get more specific. When code detects a failure condition—something exceptional or outside of the ordinary or expected—that code creates a new instance of the class **`System.Exception`** (or something derived from **`Exception`**). This exception object represents the problem that occurred, and different derived classes represent specific categories of errors. This exception object is then *thrown*, which begins the process of looking for a handler farther up the call stack. With the code above, **`Convert.ToInt32`** contains the code that detects this error, creates the exception, and throws it. We will soon see how to do that ourselves.

The program will first look in the **`Convert.ToInt32`** method for an appropriate **`catch`** block that handles this error. It does not exist, so the search continues to the calling method, which is our code. If our code did not have a **`catch`** block that could handle the issue, the search would continue even further upward until an appropriate **`catch`** block handler is found or it escapes the program's entry point, in which case, the program would end in a crash.

Fortunately, this code now handles such errors, so the search ends at our **`catch`** block.

Once the code within the **`catch`** block runs, execution will resume after the **`try`**/**`catch`** block.

If a **`try`** block has many statements and the first throws an exception, the rest of the code will not run. It is crucial to pick the right section of code to place in your **`try`** blocks, but smaller is usually better.

Handling Specific Exception Types

Our **`catch`** block above handles all possible exception types. That's not usually what you want. It is generally better to be more specific about the kind of error. Handle only the types you can recover from and handle different error types differently.

If we look at the documentation for **`Convert.ToInt32(string)`**, we see that it might throw a **`System.FormatException`** or a **`System.OverflowException`**. The **`FormatException`** class occurs when the text is not numeric, and **`OverflowException`** occurs when the number is too big to store in an **`int`**. It makes sense to handle these in different ways. We can modify our **`catch`** block into the following:

```
try
{
    number = Convert.ToInt32(response);
}
catch (FormatException)
{
    number = -1;
    Console.WriteLine($"I do not understand '{ response }'.");
}
catch (OverflowException)
{
    number = -1;
    Console.WriteLine($"That number is too big!");
}
```

This code defines two separate **`catch`** blocks associated with a single **`try`** block, one for each of the ways **`Convert.ToInt32`** can fail. Doing so allows us to treat each error type differently.

When looking for an exception handler, the order matters. **`FormatException`** and **`OverflowException`** are distinct exception types, but consider this code:

```
try { ... }
catch (FormatException) { ... }
catch (Exception) { ... }
```

The first block will handle a **`FormatException`** because it comes first. The second one will handle every other exception type because everything is derived from **`Exception`**.

A **`try`**/**`catch`** block does not need to handle every imaginable exception type. We could simply do the following if we wanted to:

```
try { ... }
catch (FormatException) { ... }
```

This will catch **FormatException** objects but leave other errors for something else to address. Code that cannot reasonably resolve a specific problem type should not catch it.

Using the Exception Object

An exception handler can use the exception object in its body if it needs to. To do so, add a name after the exception type in the **catch**'s parentheses:

```
try { ... }
catch (FormatException e)
{
    Console.WriteLine(e.Message);
}
```

The **Exception** class defines a **Message** property, so all exception objects have it. Other exception types may add additional data that can be helpful, though neither **FormatException** nor **OverflowException** does this.

THROWING EXCEPTIONS

Let's now look at the other side of the equation: creating and throwing new exceptions.

The first thing your code must do is recognize a problem. You will have to determine for yourself what counts as an unresolvable error in your code. But once you have detected such a situation, you are ready to create and throw an exception.

Exceptions are represented by any object whose class is **Exception** or a derived class. Creating an exception object is like making any other object: you use **new** and invoke one of its constructors. Once created, the next step is to throw the exception, which begins the process of finding a handler for it. These are often done in a single statement:

```
throw new Exception();
```

The **new Exception()** part creates the exception object. The **throw** keyword is the thing that initiates the hunt up the call stack for a handler. In context, this could look something like this:

```
Console.WriteLine("Name an animal.");
string? animal = Console.ReadLine();
if (animal == "snake") throw new Exception(); // Why did it have to be snakes?
```

The **Exception** class represents the most generic error in existence. With this code, all we know is that *something* went wrong. In general, you want to throw instances of a class derived from **Exception**, not **Exception** itself. Doing so allows us to convey what went wrong more accurately and enables handlers to be more specific about if and how to handle it.

There is a mountain of existing exception types that you can pick from, which represent various situations. Here are a few of the more common ones, along with their meanings.

Exception Name	Meaning
NotImplementedException	"The programmer hasn't written this code yet."
NotSupportedException	"I will never be able to do this."
InvalidOperationException	"I can't do this in my current state, but I might be able to in another state."
ArgumentOutOfRangeException	"This argument was too big (too small, etc.) for me to use."
ArgumentNullException	"This argument was null, and I can't work with a null value."
ArgumentException	"Something is wrong with one of your arguments."
Exception	"Something went wrong, but I don't have any real info about it."

Rather than using **`new Exception()`** earlier, we should have picked a more specific type. Perhaps **`NotSupportedException`** is a better choice:

```
Console.WriteLine("Name an animal.");
string? animal = Console.ReadLine();
if (animal == "snake") throw new NotSupportedException();
```

Most exception types also allow you to supply a message as a parameter, and it is often helpful to include one to help programmers who encounter it later:

```
if (animal == "snake") throw new NotSupportedException("I have ophidiophobia.");
```

Depending on the exception type, you might be able (or even required) to supply additional information to the constructor.

If one of the existing exception types isn't sufficient to categorize an error, make your own by defining a new class derived from **`Exception`** or another exception class:

```
public class SnakeException : Exception
{
    public SnakeException() { }
    public SnakeException(string message) : base(message) { }
}
```

Always use a meaningful exception type when you throw exceptions. Avoid throwing plain old **`Exception`**. Use an existing type if it makes sense. Otherwise, create a new one.

THE FINALLY BLOCK

A **`finally`** block is often used in conjunction with **`try`** and **`catch`**. A **`finally`** block contains code that should run regardless of how the flow of execution leaves a **`try`** block, whether that is by typical completion of the code, throwing an exception, or an early return:

```
try
{
    Console.WriteLine("Shall we play a game?");
    if (Console.ReadLine() == "no") return;

    Console.WriteLine("Name an animal.");
    string? animal = Console.ReadLine();
    if (animal == "snake") throw new SnakeException();
}
catch (SnakeException) { Console.WriteLine("Why did it have to be snakes?"); }
finally
{
    Console.WriteLine("We're all done here.");
}
```

There are three ways to exit the **try** block above; the **finally** block runs in all of them. If the early return on line 4 is encountered, the **finally** block executes before returning. If the end of the **try** block is reached through normal execution, the **finally** block is executed. If a **SnakeException** is thrown, the **finally** block executes after the **SnakeException** handler runs. If this code threw a different exception not handled here, the **finally** block still runs before leaving the method to find a handler.

The purpose of a **finally** block is to perform cleanup reliably. You know it will always run, so it is a good place to put code that ensures things are left in a good state before moving on. As such, it is not uncommon to have just a **try** and a **finally** with no **catch** blocks at all.

Exception Guidelines

Let's look at some guidelines for throwing and catching exceptions.

What to Handle

Any exception that goes unhandled will crash the program. In general, this means you should have a bias for catching exceptions instead of letting them go. But exception handling code is more complicated than code that does not. Code understandability is also valuable.

Catching exceptions is especially important in products where failure means loss of human life or injury versus a low-stakes utility that will almost always be used correctly. In these low-stakes, low-risk programs, skipping some or all the exception handling could be an acceptable choice. Every program we have made so far could arguably fit into this category.

Still, handling exceptions allows a program to deal with strangeness and surprises. Code that does this is *robust*. Even if nobody dies from a software crash, your users will appreciate it being robust. With exception handling knowledge, you should have a bias for doing it, not skipping it.

Only Handle What You Can Fix

If an exception handler cannot resolve the problem represented by the exception, the handler should not exist. Instead, the exception should be allowed to continue up the call stack, hoping that something farther up has meaningful resolution steps for the problem. This is a counterpoint to the previous item. If there is no recourse for an error, it is reasonable for the program to end.

There are some allowances here. Sometimes, a handler will repair or address what it can (even just logging the problem) while still allowing the exception to continue (described later).

Use the Right Exception Type

An exception's type (class) is the simplest way to differentiate one error category from another. By picking the right exception type when throwing exceptions (making your own if needed), you make life easier when handling exceptions.

Avoid Pokémon Exception Handling

Sometimes, it is tempting to handle any possible error in the same way with this:

```
catch (Exception) { Console.WriteLine("Something went wrong."); }
```

Some programmers call this Pokémon exception handling. Using **`catch (Exception)`** catches every possible exception with no… um… exceptions. It is reminiscent of the catchphrase from the game Pokémon, "Gotta catch 'em all!"

The problem with treating everything the same is that it is often too generic. "Something went wrong" is an awful error message. Whether solved by humans or code, an error's recourse is rarely the same for all possible errors.

There are, of course, times where this is the only thing that makes sense. Some people will put a **`catch (Exception)`** block around their entire program to catch any stray unhandled exceptions as the program is dying to produce an error report or something similar. But letting the program attempt to resume is often dangerous because we have no guarantees about the program's state when the exception occurred. So use Pokémon exception handling sparingly, and in general, let the program die afterward.

Avoid Eating Exceptions

A **`catch`** block that looks like this is usually bad:

```
catch (SomeExceptionType) { }
```

An empty handler like this is referred to as "eating the exception," "swallowing the error," or "failing silently." Correct exception handling rarely requires doing nothing at all. Empty catch blocks nearly always represent a programmer who got lazy.

The problem is that an error occurred, and no response was taken to address it. It may leave the program in a weird or inconsistent state—one in which the program should not be running.

Eating exceptions is especially bad when combined with the previous item: **`catch (Exception) { }`**. Here, every single error is caught and thrown right into the garbage chute.

Avoid Throwing Exceptions When Possible

Exceptions are a useful tool, but you should not throw exceptions that you do not need to throw. Avoid exceptions if simple logic is sufficient. The following is trivialized but illustrative:

```
try
{
    Console.WriteLine("Name an animal.");
    string? animal = Console.ReadLine();
    if (animal == "snake") throw new Exception();
}
catch (Exception)
{
    Console.WriteLine("Snakes. Why did it have to be snakes?");
}
```

Instead of that, just do this:

```
Console.WriteLine("Name an animal.");
string? animal = Console.ReadLine();
if (animal == "snake") Console.WriteLine("Why did it have to be snakes?");
```

The result is the same, but with cleaner code. If you can use logic like **`if`** statements, loops, etc., those are usually better approaches.

Come Back with Your Shield or On It

In ancient Greece, soldiers would go into battle advised to "come back with your shield or on it." Coming back with your shield meant winning the fight. Coming back on your shield meant dying with honor and being carried home on your shield. Somebody who abandons their duty would run from the battle and drop their heavy shield in the process, returning home alive but without their shield. Or so the story goes. Military strategy aside, this is a good rule for exceptions. When a method runs, it should either do its job and run to completion or fail with honor by throwing an exception but leaving things in the state it began in. It should not abandon its job halfway through and leave things partly changed and partly unchanged.

A **`finally`** block is often your best tool for ensuring you can get back to your original state.

If you cannot put things back exactly as they were when you started, you should at least put things into a self-consistent state. To illustrate, consider this contrived scenario. You have a variable that is expected to be even but must be incremented twice and may throw an exception while changing it:

```
_evenNumber++;
MaybeThrowException();
_evenNumber++;
```

If **`_evenNumber`** was a **`4`** and things go well, this will become a **`6`** afterward. If an exception is thrown, then using the "with your shield or on it" rule (also called the *strong exception guarantee*), you should revert **`_evenNumber`** to a **`4`**. In this case, it requires extra bookkeeping:

```
int startingValue = _evenNumber;
try
{
    _evenNumber++;
    MaybeThrowException();
    _evenNumber++;
}
finally
{
    if (_evenNumber % 2 != 0) _evenNumber = startingValue;
}
```

If that is not possible, we should not leave **`_evenNumber`** as a **`5`**, which is an odd number and goes against expectations. Setting **`_evenNumber`** to **`0`** in a **`finally`** block at least leaves the program in a "correct" state.

```
try
{
    _evenNumber++;
    MaybeThrowException();
    _evenNumber++;
}
finally
{
    if (_evenNumber % 2 != 0) _evenNumber = 0;
}
```

Advanced Exception Handling

SIDE QUEST

In this section, we will visit a handful of more advanced aspects of exception handling. Each of these is an essential feature that sees a fair bit of use.

Stack Traces

Each exception, once thrown, contains a *stack trace*. The stack trace describes methods currently on the stack, from the program's entry point to the exception's origination site. Consider this simple program:

```
DoStuff();

void DoStuff() => DoMoreStuff();
void DoMoreStuff() => throw new Exception("Something terrible happened.");
```

The main method calls **`DoStuff`**, which calls **`DoMoreStuff`**, which throws an exception. The stack trace for this exception reveals that the exception occurred in **`DoMoreStuff`**, called by **`DoStuff`**, called by **`Main`**.

Each exception has a **`StackTrace`** property that you can use to see this stack trace. However, **`Exception`** has overridden **`ToString`** to include this. Doing something like **`Console.WriteLine(e)`** is an easy way to see it. To illustrate, we can wrap **`DoStuff`** in a **`try`**/**`catch`** block and use the console window to display the exception:

```
try { DoStuff(); }
catch (Exception e) { Console.WriteLine(e); }
```

Running this displays the following:

```
System.Exception: Something terrible happened.
   at Program.<<Main>$>g__DoMoreStuff|0_1() in C:\some\path\Program.cs:line 14
   at Program.<<Main>$>g__DoStuff|0_0() in C:\some\path\Program.cs:line 12
   at Program.<Main>$(String[] args) in C:\some\path\Program.cs:line 7
```

This gives you the exception type and message, followed by the stack trace. Each element in the stack makes an appearance, showing the method signature, the path to the file, and even the line number!

This particular stack trace is short but uglier than most. The compiler names your main method **`<Main>$`**, and local functions like **`DoStuff`** and **`DoMoreStuff`** always end up with strange final names. Most stack traces you see will not be so alien.

The stack trace can help you understand what happened and where things went wrong. Having said that, if you are running your program from Visual Studio (or another IDE), the debugger can also show this information and more. See Bonus Level C for more information.

Rethrowing Exceptions

After catching an exception, you sometimes realize that you cannot handle the exception after all and need it to continue up the call stack. A simple approach is just to throw it again:

```
try { DoStuff(); }
catch (Exception e)
```

```
{
    Console.WriteLine(e);
    throw e;
}
```

There is a catch. An exception's stack trace is updated when thrown, not when created. That means when you throw an exception, as shown above, the stack trace will change to its new location in this **`catch`** block, losing useful information. There are times where this is desirable. Most of the time, it is not. There's another option:

```
try { DoStuff(); }
catch (Exception e)
{
    Console.WriteLine(e);
    throw;
}
```

A bare **`throw;`** statement will rethrow the caught exception without modifying its original stack trace. This makes it easy to let a caught exception continue looking for a handler.

Perhaps the more useful case for rethrowing exceptions is to inject some logic for an exception *without* handling or resolving it. The code above does just that by logging (to the console window) exceptions as they occur without preventing the crash.

Inner Exceptions

Sometimes, when you catch an exception, you want to replace it with another. This is especially common when some low-level thing is misbehaving, and you want to transform it into a set of exception types that indicate higher-level problems. You can, of course, catch the low-level exception and then throw a new exception:

```
try { DoStuff(); }
catch (FormatException e)
{
    throw new InvalidDataException("The data must be formatted correctly.");
}
catch (NullReferenceException e)
{
    throw new InvalidDataException("The data is missing.");
}
```

Like with rethrowing exceptions, this loses information in the process. Each exception has a property called **`InnerException`**, which can store another exception that may have been the underlying cause.

Most exception classes let you create new instances with no parameters (**`new Exception()`**), with a single message parameter (**`new Exception("Oops")`**), or with a message and an inner exception (**`new Exception("Ooops", otherException)`**). This inner exception allows you to supply an underlying cause when creating a new exception, preserving the root cause. When you create new exception types, you should make similar constructors in your new class to allow the pattern to continue.

Exception Filters

Most of the time, you decide whether to handle an exception based solely on the exception's type. If you need more nuance, you can use exception filters. An exception filter is a simple **`bool`** expression that must be true for a **`catch`** block to be selected. The filter allows you to inspect the exception object's other properties. The following uses a made-up **CodedErrorException**:

```
try { DoStuff(); }
catch (CodedErrorException e) when (e.ErrorCode == ErrorCodes.ConnectionFailure)
{ ... }
```

This **catch** block will only execute for **CodedErrorException**s whose **ErrorCode** property is **ErrorCodes.ConnectionFailure**.

Challenge Exepti's Game 100 XP

On the Island of Exceptions, you find the village of Excepti, which has seen little happiness and joy since the arrival of The Uncoded One. The Exceptians used to have a game that they played called Cookie Exception. The village leader, Noit Pecxe, promises the warriors of Excepti will join you against the Uncoded One if you can recreate their ancient tradition in a program. Noit offers you the Medallion of Exceptions as well.

Cookie Exception is played by gathering nine chocolate chip cookies and one oatmeal raisin cookie. The cookies are mixed and put in a dark room with two players who can't see the cookies. Each player takes a turn picking a cookie randomly and shoving it in their mouth without seeing whether it is a delicious chocolate chip cookie or an awful oatmeal raisin cookie. If they pick wrong and eat the oatmeal raisin cookie, they lose. If their opponent eats the oatmeal raisin cookie, then they win.

Objectives:

- The game will pick a random number between 0 and 9 (inclusive) to represent the oatmeal raisin cookie.
- The game will allow players to take turns picking numbers between 0 and 9.
- If a player repeats a number that has been already used, the program should make them select another. **Hint:** If you use a **List<int>** to store previously chosen numbers, you can use its **Contains** method to see if a number has been used before.
- If the number matches the one the game picked initially, an exception should be thrown, terminating the program. Run the program at least once like this to see it crash.
- Put in a **try**/**catch** block to handle the exception and display the results.
- **Answer this question:** Did you make a custom exception type or use an existing one, and why did you choose what you did?
- **Answer this question:** You could write this program without exceptions, but the requirements demanded an exception for learning purposes. If you didn't have that requirement, would you have used an exception? Why or why not?

LEVEL 36

DELEGATES

Speedrun

- A delegate is a variable that stores methods, allowing them to be passed around like an object.
- Define delegates like this: **`public delegate float NumberDelegate(float number);`**. This identifies the return type and parameter list of the new delegate type.
- Assign values to delegate variables like this: **`NumberDelegate d = AddOne;`**
- Invoke the method stored in a delegate variable: **`d(2)`**, or **`d.Invoke(2)`**.
- **`Action`**, **`Func`**, and **`Predicate`** are pre-defined generic delegate types that are flexible enough that you rarely have to build new delegate types from scratch.
- Delegates can refer to multiple methods if needed, and each method will be called in turn.

DELEGATE BASICS

A *delegate* is a variable that holds a reference to a method or function. This feature allows you to pass around chunks of executable code as though it were simple data. That may not seem like a big deal, but it is a game-changer. Delegates are powerful in their own right but also serve as the basis of many other powerful C# features.

Let's look at the type of problem they help solve. Suppose you have this method, which takes an array of numbers and produces a new array where every item has been incremented. If the array **`[1, 2, 3, 4, 5]`** is passed in, the result will be **`[2, 3, 4, 5, 6]`**.

```
int[] AddOneToArrayElements(int[] numbers)
{
    int[] result = new int[numbers.Length];

    for (int index = 0; index < result.Length; index++)
        result[index] = numbers[index] + 1;

    return result;
}
```

What if we also need a method that subtracts one instead? Not a big deal:

```
int[] SubtractOneFromArrayElements(int[] numbers)
{
    int[] result = new int[numbers.Length];

    for (int index = 0; index < result.Length; index++)
        result[index] = numbers[index] - 1;

    return result;
}
```

These two methods are identical except for the code that computes the new array's value from the original value. You could create both methods and call it a day, but that is not ideal. It is a large chunk of duplicated code. If you needed to fix a bug, you'd have to do so in two places.

We could maybe add another parameter to indicate how much to change the number:

```
int[] ChangeArrayElements(int[] numbers, int amount)
{
    int[] result = new int[numbers.Length];

    for (int index = 0; index < result.Length; index++)
        result[index] = numbers[index] + amount;

    return result;
}
```

To add and subtract, we could call **`ChangeArrayElements(numbers, +1)`** and **`ChangeArrayElements(numbers, -1)`**. But there is only so much flexibility we can get. What if we wanted a similar method that doubled each item or computed each item's square root?

To give the calling method the most flexibility, we can ask it to supply a method to use instead of adding a specific number.

This is easier to illustrate with an example. Let's start by defining the methods **`AddOne`**, **`SubtractOne`**, and **`Double`**:

```
int AddOne(int number) => number + 1;
int SubtractOne(int number) => number - 1;
int Double(int number) => number * 2;
```

These methods have the same parameter list (a single **`int`** parameter) and the same return type (also an **`int`**). That similarity is essential; it is what will make them interchangeable.

The next step is for us to give a name to this pattern by defining a delegate type:

```
public delegate int NumberDelegate(int number);
```

This defines a new type, like defining a new enumeration or class. Defining a new delegate type requires a return type, a name, and a parameter list. In this sense, it looks like a method declaration, aside from the **`delegate`** keyword.

Variables that use delegate types can store methods. But the method must match the delegate's return type and parameter types to work. A variable whose type is **`NumberDelegate`** can store any method with an **`int`** return type and a single **`int`** parameter. Lucky for us, **`AddOne`**, **`SubtractOne`**, and **`Double`** all meet these conditions. That means we can make a variable that can store a reference to any of them.

There are three parts to using a delegate: making a variable of that type, assigning it a value, and then using it.

Any variable can use a delegate for its type, just like we saw with enumerations and classes. We can make a method with a parameter whose type is **`NumberDelegate`**, which will allow callers of the method to supply a different method to invoke when the time is right:

```
int[] ChangeArrayElements(int[] numbers, NumberDelegate operation) { ... }
```

To call **`ChangeArrayElements`** with the delegate, we name the method we want to use:

```
ChangeArrayElements(new int[] { 1, 2, 3, 4, 5 }, AddOne);
```

Note the lack of parentheses! With parentheses, we'd be invoking the method and passing its return value. Instead, we are passing the method *itself* by name.

If the method is an instance method, you can name the object with its method:

```
SomeClass thing = new SomeClass();
ChangeArrayElements(new int[] { 1, 2, 3, 4, 5 }, thing.DoIt);
```

The C# compiler is smart enough to keep track of the fact that the delegate must store a reference to the instance (**`thing`**) and know which method to call (**`DoIt`**).

On rare occasions, the compiler may struggle to understand what you are doing. In these cases, you may need to be more formal with something like this:

```
ChangeArrayElements(new int[] { 1, 2, 3, 4, 5 }, new NumberDelegate(AddOne));
```

That shouldn't happen very often, though.

Let's see how **`ChangeArrayElements`** would use this delegate-typed variable. Because a delegate holds a reference to a method, you will eventually want to invoke the method. There are two ways to do this. The first is shown here:

```
int[] ChangeArrayElements(int[] numbers, NumberDelegate operation)
{
    int[] result = new int[numbers.Length];

    for (int index = 0; index < result.Length; index++)
        result[index] = operation(numbers[index]);

    return result;
}
```

You can invoke the method in a delegate variable by using parentheses. Invoking a method in a delegate-typed variable looks like a typical method call, except perhaps the capitalization. (Most methods in C# start with a capital letter. Most parameters do not.)

The second way is to use the delegate's **`Invoke`** method:

```
result[index] = operation.Invoke(numbers[index]);
```

These are the same thing for all practical purposes, though this second option allows you to check for null with a simple **`operation?.Invoke(numbers[index])`**.

By looking at this code, you can see why delegates are called that. **`ChangeArrayElements`** knows how to iterate through the array and build a new array, but it doesn't understand how to compute new values from old values. It expects somebody else to do that work, and when the time comes, it delegates that job to the delegate object.

Delegates can significantly increase the flexibility of sections of code. It can allow you to define algorithms with replaceable elements in the middle, filled in by other methods via delegates. That makes them a valuable tool to add to your C# toolbox.

THE ACTION, FUNC, AND PREDICATE DELEGATES

In the last section, we defined a new delegate type to use in our program. That has its uses—if you want a specific name given to a method pattern—but if you play your cards right, you won't have to define new delegate types often. The Base Class Library contains a flexible and extensive collection of delegate types that cover most scenarios.

Two sets of generic delegate types cover virtually all situations: **`Action`** and **`Func`**. Each is a *set* of generic delegates rather than a single delegate type.

The **`Action`** delegates have a **`void`** return type. They capture scenarios where a method performs a job without producing a result. The simplest one, known simply as **`Action`**, is a delegate for any function with no parameters and a **`void`** return type, such as **`void DoSomething()`**.

If you need one parameter, **`Action<T>`** is what you want. It is generic, so the right flavor will allow you to account for any parameter type—for example, **`Action<string>`** for a method like **`void DoSomething(string value)`**. There are versions of **`Action`** with up to 16 parameters, though if you have a method with more than 16 parameters, change your design. You could use **`Action<string, int, bool>`** for **`void DoSomething(string message, int number, bool isFancy)`**.

The **`Func`** delegates (short for "function") are for when you need a return value. **`Func<TResult>`** is the simplest version, and it has a generic return type (**`TResult`**). Use this for a method with no parameters. **`Func<int>`** could be used for the method **`int GetNumber()`**. If you need parameters, there's a **`Func`** for that too. **`Func<T, TResult>`** is for a single parameter. You could use **`Func<int, double>`** for **`double DoSomething( int number)`**. Like **`Action`**, there is a version with up to 16 parameters plus a return type, and all are generic. For example, **`Func<string, int, bool, double>`** works for **`double DoSomething(string message, int number, bool isFancy)`**.

You can use one of the above delegate types for any situation where delegates could come in handy. Our **`NumberDelegate`** could have been done with **`Func<int, int>`**. Some programmers almost exclusively use these delegate types. Others tend to make their own so they can give them more descriptive names.

One other delegate type is worth noting here: **`Predicate<T>`**. The mathematical definition of *predicate* is a condition used to determine whether something belongs to a set. **`Predicate<T>`** represents a method that takes an object of the generic type **`T`** and returns a **`bool`**. (That makes it equivalent to **`Func<T, bool>`**.) Its definition looks something like this:

```
public delegate bool Predicate<T>(T value);
```

(This also illustrates how to define generic delegates.) For example, we could define **`IsEven`** and **`IsOdd`** methods that tell you if a number belongs to the set of even numbers or odd numbers. The name **`Predicate<T>`** reveals its intended use better than **`Func<T, bool>`** and spares you from filling in two generic type parameters.

MulticastDelegate and Delegate Chaining

SIDE QUEST

Behind the scenes, declaring new delegate types creates new classes derived from the special class **`MulticastDelegate`**. That name hints at doing things in multiples, and indeed you can. Each delegate object can store many methods, not just a single one. This collection is called a *delegate chain*. When a delegate is invoked, each method in the delegate chain will be called in turn.

In practice, this is rare. (Though see Level 37 where events put this ability to use.) Doing so brings up at least one notable concern: what return value does a delegate with multiple methods return? It cannot account for all of the return values. The return value will be that of the last method attached, ignoring the rest. If you are going to attach multiple methods to a multicast delegate, you should only do so with the **`void`** return type.

Attaching additional methods to a delegate can be done with the **`+=`** or **`+`** operators and subsequently detached with the **`-=`** or **`-`** operators. For example, suppose you have the following delegate:

```
public delegate void Log(LogLevel level, string message);
```

You could get a delegate-typed variable to invoke many methods with the same parameter list and return type like this:

```
Log logMethods = LogToConsole;
logMethods += LogToDatabase;
logMethods += LogToTextFile;
```

When you invoke **`logMethods(LogLevel.Warning, "A problem happened.");`**, it will call all three of those methods. You could also write it like this:

```
Log logMethods = new Log(LogToConsole) + LogToDatabase + LogToTextFile;
```

If any of the methods throw an exception while running, the other delegate methods will not get a chance to run. When used this way, attached methods should not let exceptions escape.

Challenge **The Sieve** **100 XP**

The Island of Delgata is home to the Numeromechanical Sieve, a machine that takes numbers and judges them as good or bad numbers. In ancient times, the sieve could be supplied with a single method to use as a filter by the island's rulers, making the sieve adaptable as leadership changed over time. The Delgatans will give you the Medallion of Delegates if you can reforge their Numeromechanical Sieve.

Objectives:

- Create a **`Sieve`** class with a **`public bool IsGood(int number)`** method. This class needs a constructor with a delegate parameter that can be invoked later within the **`IsGood`** method. **Hint:** You can make your own delegate type or use **`Func<int, bool>`**.
- Define methods with an **`int`** parameter and a **`bool`** return type for the following: (1) returns true for even numbers, (2) returns true for positive numbers, and (3) returns true for multiples of 10.
- Create a program that asks the user to pick one of those three filters, constructs a new **`Sieve`** instance by passing in one of those methods as a parameter, and then ask the user to enter numbers repeatedly, displaying whether the number is good or bad depending on the filter in use.
- **Answer this question:** Describe how you could have also solved this problem with inheritance and polymorphism. Which solution seems more straightforward to you, and why?

LEVEL 37

EVENTS

Speedrun

- Events allow a class to notify interested observers that something has occurred, allowing them to respond to or handle the event: **`public event Action ThingHappened;`**
- Events use a delegate type to indicate what a handler must look like.
- Raise events like this: **`ThingHappened()`**, or **`ThingHappend?.Invoke();`**
- Events can use any delegate type but should avoid non-**`void`** return types.
- Other types can subscribe and unsubscribe to an event by providing a method: **`something.ThingHappened += Handler;`** and **`something.ThingHappened -= Handler;`**
- Don't forget to unsubscribe; objects that stay subscribed will not get garbage collected.

C# EVENTS

In C#, events are a mechanism that allows an object to notify others that something has changed or happened so they can respond.

Suppose we were making the game of *Asteroids*. Let's say we have a **`Ship`** class, representing the concept of a ship, including if it is dead or alive, and a **`SoundEffectManager`** class, which has the responsibility to play sounds. We have an instance of each. When a ship blows up, an explosion sound should play. We have a few options for addressing this.

If the **`Ship`** class knows about the **`SoundEffectManager`**, it could call a method: **`_soundEffectManager.PlaySound("Explosion");`**. This design is not unreasonable. But if eight things need to respond to the ship exploding, it's less reasonable for **`Ship`** itself to reach out and call all of those different methods in response. As the number grows, the design looks worse and worse.

Alternatively, we could ask each of those objects to implement some interface like this:

```
public interface IExplosionHandler
{
    void HandleShipExploded();
}
```

SoundEffectManager could implement this interface and play the right sound. The other seven objects could do a similar thing. The **Ship** class can have a list of **IExplosion Handler** objects and call their **HandleShipExploded** method after the ship explodes. A slice of **Ship** might look like this:

```
public class Ship
{
    private List<IExplosionHandler> _handlers = new List<IExplosionHandler>();

    public void AddExplosionHandler(IExplosionHandler newHandler) =>
                _handlers.Add(newHandler);

    private void TellHandlersAboutExplosion()
    {
        foreach (IExplosionHandler handler in _handlers)
            handler.HandleShipExploded();
    }
}
```

Something within **Ship** would need to recognize that the ship has exploded and call **TellHandlersAboutExplosion**.

The nice part of this setup is that the ship does not need to know all eight handlers' unique aspects. Those objects sign up to be notified by calling **AddExplosionHandler**.

C# provides a mechanism based on this approach that makes things very easy: *events*. Any class can create an event as a member, similar to making properties and methods. Any other object interested in reacting to the event—a *listener* or an *observer*—can subscribe to the event to be notified when the event occurs. The class that owns the event can then *raise* or *fire* the event when the time is right, causing each listener's handler to run.

Defining an event is shown below:

```
public class Ship
{
    public event Action? ShipExploded;

    // The rest of the ship's members are defined here.
}
```

An event is defined using the **event** keyword, followed by a delegate type, then its name. Like every other member type, you can add an accessibility modifier to the event, as we did here with **public**. Events are typically public.

In many ways, declaring an event is like an auto-property. Behind the scenes, a delegate object is created as a backing field for this event. In the case above, this delegate's type will be **Action?** (no parameters and a **void** return type, and with **null** allowed) since that is what the event's declaration named.

When the **Ship** class detects the explosion, it will raise or fire this event, as shown below:

```
public class Ship
{
    public event Action? ShipExploded;
    public int Health { get; private set; }

    public void TakeDamage(int amount)
    {
        Health -= amount;
        if (Health <= 0)
            ShipExploded();
    }
}
```

This notifies listeners that the event occurred, allowing them to run code in response.

Alternatively, we can use the **`Invoke`** method:

```
if (Health <= 0)
    ShipExploded.Invoke();
```

Let's look at the listener's side now. If we want **`SoundEffectManager`** to respond to the ship exploding, we define a method that matches the event's delegate type. In this case, that type is **`Action`**, which has a **`void`** return type and no parameters. This method can be called whatever we want, but names starting with **`On`** or **`Handle`** are both common:

```
public class SoundEffectManager
{
    private void OnShipExploded() => PlaySound("Explosion");

    public void PlaySound(string name) { ... }
}
```

Next, we need to attach this method to the event. We could do this in the constructor:

```
public SoundEffectManager(Ship ship)
{
    ship.ShipExploded += OnShipExploded;
}
```

This *attaches* or *subscribes* the **`OnShipExploded`** method to the event, ensuring it will be called when the event fires.

When you are done, you can *detach* or *unsubscribe* the event like this:

```
ship.ShipExploded -= OnShipExploded;
```

The benefits of events are substantial. The object declaring the event does not have to know details about each object that responds to it. Each handler subscribes to the event with one of its methods, and everything else is taken care of automatically. Plus, unlike our interface approach, objects responding to the event do not need to implement any particular interface. They can call their event handler whatever makes sense for them.

Events with Parameters

The code above used **`Action`**, which has no parameters. But events can supply data as arguments to their listeners by using a delegate type that includes parameters. For example, we could report the explosion's location with this:

```
public class Ship
{
    public event Action<Point>? ShipExploded;
    public int Health { get; private set; }
    public Point Location { get; private set; } = new Point(0, 0);

    public void TakeDamage(int amount)
    {
        Health -= amount;
        if (Health <= 0)
            ShipExploded(Location);
    }
}
```

With this, an observer would subscribe using a method with a **`Point`** parameter. It can then use that parameter in deciding how to respond.

```
public class SoundEffectManager
{
    private void OnShipExploded(Point location) =>
                    PlaySound("Explosion", CalculateVolume(location));

    public SoundEffectManager(Ship ship)
    {
        ship.ShipExploded += OnShipExploded;
    }

    public void PlaySound(string name, float volume) { ... }
    private float CalculateVolume(Point location) { ... }
    // ...
}
```

Null Events

An event may be null if nothing has subscribed to it. Our earlier examples have ignored this possibility, which is dangerous. We should either check to see if the event is null or ensure that it isn't ever null by always giving it at least one event handler. The first option is more common and can be done by simply checking for null before raising the event:

```
if (Health <= 0)
    ShipExploded?.Invoke();
```

The second option is tricky: ensure the event always has at least one handler. We rarely know of a valid handler when the object is created. That comes later. If we want to ensure the event is never null, we'll need to add a dummy do-nothing handler. You could imagine making a private **`DoNothing`** method within the class, but that's not very elegant. The more common alternative is to use a lambda expression—the topic of Level 38. I'll show you that here, even though it won't make sense yet:

```
public event Action ShipExploded = () => { }; // Uses lambdas from Level 38.
```

This initializer ensures **`ShipExploded`** will not be null, and we can change the event's type from **`Action?`** to **`Action`**. It comes with a cost: this empty method will run every time the event is raised.

In my experience, more people will just allow the event to be null and then check for null when raising the event. But this second approach still comes up.

Event Leaks

As we saw in Level 14, the garbage collector is usually great at cleaning up heap objects when they are no longer usable. Any object that is referenced by another will stay alive. That has consequences for events. The delegate backing an event will hold a reference to any object subscribed to it. That means an object can survive even if the only thing hanging on to it is an event subscription. Usually, if something is meant to be alive, something besides an event will also have a reference to it. If an object is accidentally surviving because of an event subscription alone, it is called an *event leak* or an *event memory leak*.

When an object is at the end of its life, it must unsubscribe from any events it is subscribed to, or it will not be garbage collected. (At least not until the object with the event dies as well.)

There are a lot of ways to approach this. One way—a rather poor way—is to ignore it. It only truly matters if you begin running out of memory or if all the excess event handling makes your program run slow. For tiny, short-lived programs, it may not present a big problem. It is safer to handle it, but sometimes the cost of getting it right is not worth the trouble.

A common solution is to make a **`Cleanup`** method (or pick your favorite name) that unsubscribes from any previously subscribed events. When it is time for the object to die, call the **`Cleanup`** method.

A slight variation on that idea is to name that method **`Dispose`** and make your object implement **`IDisposable`**. This is a topic covered in a bit more depth in Level 47. Several C# mechanisms will automatically call such a **`Dispose`** method, but you are still on the hook to call it yourself in other situations.

EventHandler and Friends

Using the various **`Action`** delegates with events is common, but another common choice is **`EventHandler`** (**`System`** namespace), which is defined approximately like this:

```
public delegate void EventHandler(object sender, EventArgs e);
```

It has two arguments. **`sender`** is the source of the event. This parameter makes it easy for subscribers to hook up their handler to many source objects while still telling which one raised the event. **`EventArgs`** provides additional data about the event. Strictly speaking, **`EventArgs`** does almost nothing. The only thing it defines beyond **`object`** is a static **`EventArgs.Empty`** object to be used when there is no meaningful additional data for the event. However, **`EventArgs`** is intended to be used as the base class for more specialized classes. Classes derived from **`EventArgs`** can include other data relevant to an event.

Alternatively, **`EventHandler<TEventArgs>`** is a generic delegate that allows you to require a specific **`EventArgs`**-derived class. If you always expect a specific **`EventArgs`**-based class, this will ensure you get the types right.

To use this, start by defining your own class derived from **`EventArgs`**. For example:

```
public class ExplosionEventArgs : EventArgs
{
    public Point Location { get; }
    public ExplosionEventArgs(Point location) => Location = location;
}
```

Change your event to use this new class:

```
public event EventHandler<ExplosionEventArgs>? ShipExploded;
```

Then raise the event with the current object and an appropriate **`EventArgs`** object:

```
ShipExploded?.Invoke(this, new ExplosionEventArgs(Location));
```

The observer waiting for the event would subscribe with a method matching this delegate and can use both of these arguments to make decisions:

```
private void OnShipExploded(object sender, ExplosionEventArgs args)
{
   if (sender is Ship) PlaySound("Explosion", CalculateVolume(args.Location));
   else if (sender is Asteroid) PlaySound("Pop", CalculateVolume(args.Location));
}
```

Some C# programmers prefer **`Action`**. Others prefer **`EventHandler`**. Others tend to write new delegate types and use those. Others mix and match. Any can do the job, so choose the flavor that works best for your situation.

Custom Event Accessors

SIDE QUEST

I said earlier that events are like auto-properties around an automatic delegate backing field. With properties, when you need more control than an auto-property provides, you can use a normal property and define your own getter and setter. The same can be done with events, though it is somewhat rare. You can define what subscribing and unsubscribing mean for any given event. The simplest version looks like this:

```
private Action? _shipExploded; // The backing field delegate.

public event Action ShipExploded
{
    add { _shipExploded += value; }
    remove { _shipExploded -= value; }
}
```

The **`add`** part defines what happens when something subscribes. The **`remove`** part defines what happens when something unsubscribes.

The above code does nothing that the automatic event doesn't do but opens the pathway to doing other things. For example, you could record when somebody subscribes or unsubscribes to an event. Or you could take the handler and attach it to several delegates.

With a custom event, you cannot raise the event directly. You must invoke the delegate behind it instead. The compiler is unwilling to guess how you expect the event to work with a custom event, so that burden lands on you.

Knowledge Check **Events** **25 XP**

Check your knowledge with the following questions:

1. **True/False.** Events allow one object to notify another when something occurs.
2. **True/False.** Any method can be attached to a specific event.
3. **True/False.** Once attached to an event, a method cannot be detached from an event.

Answers: (1) True. **(2)** False. **(3)** False

Challenge Charberry Trees 100 XP

The Island of Eventia survives by harvesting and eating the fruit of the native charberry trees. Harvesting charberry fruit requires three people and an afternoon, but two is enough to feed a family for a week. Charberry trees fruit randomly, requiring somebody to frequently check in on the plants to notice one has fruited. Eventia will give you the Medallion of Events if you can help them with two things: (1) automatically notify people as soon as a tree ripens and (2) automatically harvest the fruit. Their tree looks like this:

```
CharberryTree tree = new CharberryTree();

while (true)
    tree.MaybeGrow();

public class CharberryTree
{
    private Random _random = new Random();
    public bool Ripe { get; set; }

    public void MaybeGrow()
    {
        // Only a tiny chance of ripening each time, but we try a lot!
        if (_random.NextDouble() < 0.00000001 && !Ripe)
        {
            Ripe = true;
        }
   }
}
```

Objectives:

- Make a new project that includes the above code.
- Add a **Ripened** event to the **CharberryTree** class that is raised when the tree ripens.
- Make a **Notifier** class that knows about the tree (**Hint:** perhaps pass it in as a constructor parameter) and subscribes to its **Ripened** event. Attach a handler that displays something like “A charberry fruit has ripened!” to the console window.
- Make a **Harvester** class that knows about the tree (**Hint:** like the notifier, this could be passed as a constructor parameter) and subscribes to its **Ripened** event. Attach a handler that sets the tree’s **Ripe** property back to false.
- Update your main method to create a tree, notifier, and harvester, and get them to work together to grow, notify, and harvest forever.

LEVEL 38

LAMBDA EXPRESSIONS

Speedrun

- Lambda expressions let you define short, unnamed methods using simplified inline syntax: `x => x < 5` is equivalent to `bool LessThanFive(int x) => x < 5;`
- Multiple and zero parameters are also allowed, but require parentheses: `(x, y) => x*x + y*y` and `() => Console.WriteLine("Hello, World!");`
- Types can usually be inferred, but you can explicitly state types: `(int x) => x < 5`
- Lambdas have a statement form if you need more than just an expression: `x => { bool lessThan5 = x < 5; return lessThan5; }`
- Lambda expressions can use variables that are in scope at the place where they are defined.

LAMBDA EXPRESSION BASICS

C# provides a way to define small unnamed methods using a short syntax called a *lambda expression*. To illustrate where this could be useful, consider this method to count the number of items in an array that meet some condition. The condition is configurable, determined by a delegate:

```
public static int Count(int[] input, Func<int, bool> countFunction)
{
    int count = 0;

    foreach (int number in input)
        if (countFunction(number))
            count++;

    return count;
}
```

(In Level 42, we will see that all **`IEnumerable<T>`**'s have a **`Count`** method like this, so you do not usually have to write your own.)

We saw similar methods when we first learned about delegates in Level 36. We know we can call a method like this by passing in a named method:

```
int count = Count(numbers, IsEven);
```

But let's look at that **`IsEven`** method:

```
private static bool IsEven(int number) => number % 2 == 0;
```

That method is not long, but it has a lot of pomp and formality for a method that may only be used once. We can alternatively define a lambda expression right in the spot where it is used:

```
int count = Count(numbers, n => n % 2 == 0);
```

This lambda expression replaces the definition of **`IsEven`** entirely. You can see some similarities to methods with an expression body. They both use the **`=>`** operator. This operator is sometimes called the *arrow operator* or the *fat arrow operator* but is also frequently called the *lambda operator*. (In fact, lambda expressions came before expression-bodied methods!)

Yet, many of the other elements of this definition are gone. No **`private`**. No **`static`**. No stated return type. No name. No parentheses around the parameters. No type listed for the parameter. Plus, we used the variable name **`n`** instead of **`number`**.

A lambda expression defines a single-use method inline, right where it is needed. To prevent the code from getting ugly, everything in a lambda uses a minimalistic form:

- The accessibility level goes away because you cannot reuse a lambda expression elsewhere.
- The compiler infers the return type and parameter types from the surrounding context. Since the **`countFunction`** parameter is a **`Func<int, bool>`**, it is easy for the compiler to infer that **`n`** must be an **`int`**, and the expression must return a **`bool`**.
- The name is gone because it is a single-use method and does not need to be used again.
- The parentheses are gone just to make the code simpler.

Using the name **`n`** instead of **`number`** also makes the code shorter. Generally, more descriptive names are better, but C# programmers tend to use concise names in a lambda expression. When a variable is only used in the following few characters, the downsides of a short name are not nearly as significant as they are in a 30-line method.

We can do some pretty cool things with little code using a combination of lambda expressions and delegates. This counts the number of positive integers:

```
int positives = Count(numbers, n => n > 0);
```

This counts positive three-digit integers:

```
int  threeDigitCount = Count(numbers, n => n >= 100 && n < 1000);
```

Lambda expressions are different enough from normal methods that it may require some time to adjust. But with a bit of practice, you will find them a simple but powerful tool.

The Origin of the Name *Lambda*

You may be wondering why this is called a lambda expression. The name comes from lambda calculus. Lambda calculus is a type of function-oriented math—almost a mathematical programming language. The nature of lambda expressions and delegates is heavily inspired by lambda calculus. In lambda calculus, the name *lambda* comes from its usage of the Greek letter lambda (λ).

Multiple and Zero Parameters

Our lambda expressions so far have all had a single parameter. Let's talk about lambda expressions with zero or many parameters. When you have zero or multiple parameters, the parentheses come back. A lambda expression with two parameters looks like this:

```
(a, b) => a + b
```

A lambda expression with no parameters looks like this:

```
() => 4
```

These two cases *require* parentheses, but parentheses are always an option:

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

When Type Inference Fails

Sometimes, the compiler cannot infer the parameter types in a lambda expression. If you encounter this, you can name the types explicitly, as you might for a normal method:

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

Or:

```
(string a, string b) => a + b
```

If the compiler can't correctly infer the return type of a method, you can write out the return type before the parentheses that contain the parameters like this:

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

Or:

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

In all of these cases, parentheses are required.

Discards

Lambdas are often used in places where the code demands certain parameters but where you may not need all of them. If so, you can use discards for those parameters with either of the following two forms:

```
(_, _) => 1
(int _, int _) => 1
```

Lambda Statements

Most of the time, when you want a simple single-use method, an expression is all you need, and lambda expressions are a good fit. You can use a *lambda statement* in the rare cases where a statement or several statements are required. Lambda expressions and lambda statements are both sometimes referred to by the shorter catch-all name *lambda*.

Making a statement lambda is simple enough. Replace the expression body with a block body:

```
Count(numbers, n => { return n % 2 == 0; });
```

Or this:

```
Count(numbers, n => { Console.WriteLine(n); return n % 2 == 0; });
```

In these cases, both the curly braces and **return** keyword (if needed) are added back in.

As your statement lambdas grow longer, you should also consider a simple private method or a local function instead. Long lambdas complicate the line of code they live in, and as they get longer, they also get more deserving of a descriptive name.

CLOSURES

SIDE QUEST

Lambdas and local functions can do something normal methods can't do. Consider this code:

```
int threshold = 3;
Count(numbers, x => x < threshold);
```

The lambda expression has one parameter: **x**. However, it can use the local variables of the method that contains it. Here, it uses **threshold**. Lambda expressions and local functions can *capture* variables from their environment. A method plus any captured variables from its environment is called a *closure*. The ability to capture variables is a mechanism that gives lambdas more power than a traditional method.

However, it is essential to note that this captures the local variables themselves, not just their values. If those variables change over time, you may be surprised by the behavior:

```
Action[] actionsToDo = new Action[10];

for (int index = 0; index < 10; index++)
    actionsToDo[index] = () => Console.WriteLine(index);

foreach (Action action in actionsToDo)
    action();
```

This stores ten **Action** delegates, each containing a delegate that refers to a lambda expression. Each one displays the contents of **index**. Like declaring any other method, the act of declaring the lambda expression does not run it immediately. In this case, it isn't run until the **foreach** loop, where the delegates execute. Each delegate captured the **index** variable. You might expect this code to display the numbers 0 through 9. In actuality, this code displays **10** ten times. By the time the lambdas runs, **index** has been incremented to 10.

You can address this by storing the value in a local variable that never changes and letting the lambda capture this other variable instead:

```
for (int index = 0; index < 10; index++)
{
    int temp = index;
    actionsToDo[index] = () => Console.WriteLine(temp);
}
```

Remember that **temp**'s scope is just within the **for** loop. Each iteration through the loop will get its own variable, independent of the other passes through the loop.

As you can probably guess, the compiler is doing a lot of work behind the scenes to make captured variables and closures work. The compiler artificially extends the lifetime of those **temp** variables to allow them to stay around until the capturing delegate is cleaned up.

You can also capture variables and use closures with local functions. And remember, the methods you define with top-level statements, outside of any type, are local functions, which means such methods could technically use the variables in your main method.

While closures are very powerful, be careful about capturing variables that change over time. It almost always results in behavior you didn't intend. To prevent a lambda or local function from accidentally capturing local variables, you can add the `static` keyword to them, which causes any captured variables to become a compiler error:

```
Count(new int[] { 1, 2, 3 }, static n => { return n % 2 == 0; });
```

Knowledge Check Lambdas 25 XP

Check your knowledge with the following questions:

1. **True/False.** Lambda expressions are a special type of method.
2. **True/False.** You can name a lambda expression.
3. Convert the following to a lambda: `bool IsNegative(int x) { return x < 0; }`
4. **True/False.** Lambda expressions can only have one parameter.

Answers: (1) True. **(2)** False. **(3)** x => x < 0. **(4)** False.

Challenge The Lambda Sieve 50 XP

The city of Lambdan, also on the Island of Delgata, believes that the great Numeromechanical Sieve, which you worked on in Level 36, could be made better by using lambda expressions instead of regular, named methods. If you can help them convince island leadership to make this change, they will give you the Lambda Medallion and pledge the Lambdani Fleet's assistance in the coming final battle.

Objectives:

- Modify your *The Sieve* program from Level 36 to use lambda expressions for the constructor instead of named methods for each of the three filters.
- **Answer this question:** Does this change make the program shorter or longer?
- **Answer this question:** Does this change make the program easier to read or harder?

LEVEL 39

FILES

Speedrun

- **File-related types all live in the `System.IO` namespace.**
- **`File`** lets you read and write files: **`string[] lines = File.ReadAllLines( "file.txt"); File.WriteAllText("file.txt", "contents");`**
- **`File`** does manipulation (create, delete, move, files); **`Directory`** does the same with directories.
- **`Path`** helps you combine parts of a file path or extract interesting elements out of it. The **`File`** class is a vital part of any file I/O.
- You can also use streams to read and write files a little at a time.
- Many file formats have a library you can reuse, so you do not have to do a lot of parsing yourself.

Many programs benefit from saving information in a file and later retrieving it. For example, you might want to save settings for a program into a configuration file. Or maybe you want to save high scores to a file so that the player's previous scores remain when you close and reopen the game.

The Base Class Library contains several classes that make this easy. We will look at how to read and write data to a file in this level.

All of the classes we discuss in this level live in the **`System.IO`** namespace. This namespace is automatically included in modern C# projects, but if you're using older code, you will need to use fully qualified names or add a **`using System.IO;`** directive (Level 33).

THE FILE CLASS

The **`File`** class is the key class for working with files. It allows us to get information about a file and read and write its contents. To illustrate how the **`File`** class works, let's look at a small *Message in a Bottle* program, which asks the user for a message to display the next time the program runs. That message is placed in a file. When the program starts, it shows the message from before, if it can find one.

We can start by getting the text from the user. This uses only things familiar to us:

```
Console.Write("What do you want me to tell you next time? ");
string? message = Console.ReadLine();
```

The **`File`** class is static and thus contains only static methods. **`WriteAllText`** will take a string and write it to a file. You supply the path to the destination file, as well as the text itself:

```
Console.Write("What do you want me to tell you next time? ");
string? message = Console.ReadLine();
File.WriteAllText("Message.txt", message);
```

This alone creates a functioning program, even though it does not do everything we set out to do. If we run it, our program asks for text, makes a file called *Message.txt*, and places the user's text in it.

Where exactly does that file get created? **`WriteAllText`**—and every method in the **`File`** class that asks for a path—can work with both absolute and relative paths. An absolute path describes the whole directory structure from the root to the file. For example, I could do this to write to a file on my desktop:

```
File.WriteAllText("C:/Users/RB/Desktop/Message.txt", message);
```

A relative path leaves off most of the path and lets you describe the part beyond the current working directory. (You can also use "`..`" in a path to go up a directory from the current one in a relative path.) When your C# program runs in Visual Studio, the current working directory is in the same location as your compiled code. For example, it might be under your project folder under *\bin\Debug\net6.0\* or something similar.

If you hunt down this file, you can open it up in Notepad or another program and see that it created the file and added your text to it.

We wanted to open this file and display the last message, so let's do that with the following:

```
string previous = File.ReadAllText("Message.txt");
Console.WriteLine("Last time, you said this: " + previous);

Console.Write("What do you want me to tell you next time? ");
string? message = Console.ReadLine();
File.WriteAllText("Message.txt", message);
```

`ReadAllText` opens the named file and reads the text it contains, returning a **`string`**. The code above then displays that in the console window.

There is one problem with the code above. If we run it this way and the *Message.txt* file does not exist, it will crash. We can check to see if a file exists before trying to open it:

```
if (File.Exists("Message.txt"))
{
    string previous = File.ReadAllText("Message.txt");
    Console.WriteLine("Last time, you said this: " + previous);
}
```

That creates a more robust program that works even if the file does not exist yet.

String Manipulation

`ReadAllText` and **`WriteAllText`** are simple but powerful. You can save almost any data to a file and pull it out later with those two methods alone. You just need a way to turn what you want into a string and then parse the string to get your data back.

Let's look at a more complex problem: saving a collection of scores. Suppose we have this record:

```
public record Score(string Name, int Points, int Level);
```

And this method for creating an initial list of scores:

```
List<Score> MakeDefaultScores()
{
    return new List<Score>()
    {
        new Score("R2-D2", 12420, 15),
        new Score("C-3PO", 8543, 9),
        new Score("GONK", -1, 1)
    };
}
```

After calling this method to get our scores, how would we write all this data to a file? **`WriteAllText`** needs a string, and we have a **`List<Score>`** containing many scores.

We need a way to transform a complex object or a complex set of objects into something that can be placed into a file. This transformation is called *serialization*. The reverse is called *deserialization*. If we can serialize our scores into a string, we already know the rest.

There is no shortage of ways to serialize these scores. Here is a simple way: the CSV format. CSV, short for "comma-separated values," is a simple format that puts each item on its own line. Commas separate the item's properties. In a CSV file, our scores might look like this:

```
R2-D2,12420,15
C-3PO,8543,9
GONK,-1,1
```

`File` has a **`WriteAllLines`** method that may simplify our work. It requires a *collection* of strings instead of just one. If we can turn each score into a string, we can use **`WriteAllLines`** to get them into a file:

```
void SaveScores(List<Score> scores)
{
    List<string> scoreStrings = new List<string>();

    foreach (Score score in scores)
        scoreStrings.Add($"{score.Name},{score.Points},{score.Level}");

    File.WriteAllLines("Scores.csv", scoreStrings);
}
```

The line inside the **`foreach`** loop combines the name, score, and level into a single string, separated by commas. We do that for each score and end up with one string per score.

`File.WriteAllLines` can take it from there, so we hand it the file name and string collection, and the job is done.

Deserializing this file back to a list of scores is harder. There is a **`File.ReadAllLines`** method that is a good starting point. It returns a **`string[]`** where each string was one line in the file.

```
string[] scoreStrings = File.ReadAllLines("Scores.csv");
```

We need to take each string and chop it up to reconstitute a **`Score`** object. Since we separated data elements with commas, we can use **`string`**'s **`Split`** method to chop up the lines into its parts:

```
string scoreString = "R2-D2,12420,15";
string[] tokens = scoreString.Split(",");
```

`Split(",")` gives us an array of strings where the first item is **`"R2-D2"`**, the second item is **`"12420"`**, and the third item is **`"15"`**. If we used a **`;`** or **`|`** to separate values, we could have passed in a different argument to the **`Split`** method. Note that the *delimiter*—the character that marks the separation point between elements—is not kept when you use **`Split`** in the way shown above, but there are overloads of **`Split`** that allow that to happen.

My variable is called **`tokens`** because that is a common word for a chopped-up string's most fundamental elements.

With those elements, we can create this method to load all the scores in the file:

```
List<Score> LoadHighScores()
{
    string[] scoreStrings = File.ReadAllLines("Scores.csv");

    List<Score> scores = new List<Score>();

    foreach (string scoreString in scoreStrings)
    {
        string[] tokens = scoreString.Split(",");
        Score score = new Score(tokens[0],
                                Convert.ToInt32(tokens[1]),
                                Convert.ToInt32(tokens[2]));
        scores.Add(score);
    }

    return scores;
}
```

I should mention that the code above works most of the time but could be more robust. For example, imagine that a user enters their name as **`"Bond, James"`**. Strings can contain commas, but in our CSV file, the resulting line is **`"Bond, James,2000,16"`**. Our deserialization code will end up with four tokens and try to use **`"Bond"`** as the name and **`" James"`** as the score, which fails. We could forbid commas in player names or automatically turn commas into something else. We could also reduce the likelihood of a problem by picking a more obscure delimiter, such as **`¤`**. Few keyboard layouts can easily type that, but it is not impossible. (The official CSV format lets you put double-quote marks around strings that contain commas. This addresses the issue, but parsing that is trickier.)

Other String Parsing Methods

`File.ReadAllLines` and **`string.Split`** are enough for the above problem, but there are other string methods that you might find helpful in similar situations.

The **Trim**, **TrimStart**, and **TrimEnd** methods allow you to slice off unnecessary characters at the beginning and end of strings. The string **" Hello"** has an undesirable space character before it. **" Hello".Trim()** will produce a string without the space. It removes all whitespace from the beginning and end of the word. **TrimStart** and **TrimEnd** only trim the named side. If you want to remove another character, you can use **"$Hello".Trim('$')**. Remember that these produce new strings with the requested modification. They do not change the original string. Strings are immutable.

The **Replace** method lets you find a bit of text within another and replace it with something else. For example, if we want to turn all commas in a name to the ¤ character, we could do this: **name = name.Replace(",", "¤");**. **"Bond, James"** becomes **"Bond¤ James"**, which our parsing code can safely handle.

The **Join** method combines multiple items separated by some special string or character. We could have used this when converting **Score** objects to strings: **string.Join("|", score.Name, score.Points, score.Level);**. This method uses the **params** keyword for its second argument.

File System Manipulation

Aside from reading and writing files, the **File**, **Path**, and **Directory** class has a handful of other methods for doing file system manipulation. Let's look at those.

File has methods for copying files, moving files, and deleting files. These are all pretty self-explanatory:

```
File.Copy("Scores.csv", "Scores-Backup.csv");
File.Move("Scores.csv", "Backup/Scores.csv");
File.Delete("Scores.csv");
```

The `Directory` Class

What **File** does for files, **Directory** does for directories. (The words *directory* and *folder* are synonyms.) For example, these methods move, create, and delete a directory:

```
Directory.Move("Settings", "BackupSettings");
Directory.CreateDirectory("Settings2");
Directory.Delete("Settings2");
```

Delete requires that the directory be empty before deleting it. Otherwise, it results in an exception (**System.IO.IOException**). You could write code to remove every file in a directory yourself, but there is also an overload that allows you to force the deletion of everything inside it:

```
Directory.Delete("Settings2", true); // Careful!
```

❗ This can be extremely dangerous. You can delete entire file systems instantly with a poorly written **Directory.Delete**. Use it with extreme caution!

Directory also has several methods for exploring the contents of a directory. The names of these methods depend on whether you want results in a **string[]** (names start with **Get**) or an **IEnumerable<string>** (names start with **Enumerate**). The names also depend on whether you want files (names end with **Files**), subdirectories (names end with

Directories), or both (names end with **FileSystemEntries**). Two examples are shown below:

```
foreach (string directory in Directory.GetDirectories("Settings"))
    Console.WriteLine(directory);
foreach (string file in Directory.EnumerateFiles("Settings"))
    Console.WriteLine(file);
```

Some overloads allow you to supply a filter, enabling things like finding all files with an extension of *.txt*.

The `Path` Class

The static **Path** class has methods for working with file system paths, including combining paths, grabbing just the file name or extension from a file, and converting between absolute and relative paths. The code below illustrates all of these:

```
Console.WriteLine(Path.Combine("C:/Users/RB/Desktop/", "Settings", "v2.2"));
Console.WriteLine(Path.GetFileName("C:/Users/RB/Desktop/GrumpyCat.gif"));
Console.WriteLine(Path.GetFileNameWithoutExtension(
                                          "C:/Users/RB/Desktop/GrumpyCat.gif"));
Console.WriteLine(Path.GetExtension("C:/Users/RB/Desktop/GrumpyCat.gif"));
Console.WriteLine(Path.GetFullPath("ConsoleApp1.exe"));
Console.WriteLine(Path.GetRelativePath(".", "C:/Users/RB/Desktop"));
```

When I run these on my computer, I get the following output:

```
C:\Users\RB\Desktop\Settings/v2.2
GrumpyCat.gif
GrumpyCat
.gif
C:\Users\RB\source\repos\ConsoleApp1\ConsoleApp1\bin\Debug\net6.0\ConsoleApp1.exe
..\..\..\..\..\..\..\Desktop
```

There's More!

This is a whirlwind tour of **File**, **Directory**, and **Path**. Each has far more capabilities than we covered here, but this should give you a starting point. When you are ready, look up the documentation online or in Visual Studio's IntelliSense feature to poke around at what else these contain.

Other Ways to Access Files

The basic **ReadAllText**, **WriteAllText**, **ReadAllLines**, and **WriteAllLines** methods are a good foundation—quick and easy, without having to think too hard. But they are not the only option. Two other approaches are worth a brief discussion: streams and using a library.

Streams

The above methods require reading or writing the file all at once. Some operations are better done a little at a time. For example, let's say you're extracting millions of database entries into a CSV file. With **WriteAllText**, you would need to bring the entire dataset into memory all at once and turn it into an extremely long string to feed to **WriteAllText**. That will use a lot of memory and make the garbage collector work extremely hard. A better approach would be to grab a chunk of the data and write it to the file before continuing to the next chunk. But that requires a different approach.

We can solve this problem with *streams*. A stream is a collection of data that you typically work with a little at a time. Streams do not usually allow jumping around in the stream. They are like a conveyor belt that lets you look at each item as it goes by.

There are many different flavors of streams in the .NET world, and all of them are derived from the **`System.IO.Stream`** class. The flavor we care about here is **`FileStream`**, which reads or writes data to a file. Other stream types work with memory, the network, etc.

Streams are very low level. You can read and write bytes, and that's it. Most of the time, you want something smarter when working with a stream. This limitation is usually addressed by using another object that "wraps" the stream and provides you with a more sophisticated interface. The wrapper translates your requests to the lower level that streams require.

For example, the **`File`** class can give you a **`FileStream`** object to read or write to a file. We can then wrap a **`StreamReader`** around that to give us a better set of methods to work with than what a plain **`Stream`** or **`FileStream`** provides:

```
FileStream stream = File.Open("Settings.txt", FileMode.Open);
StreamReader reader = new StreamReader(stream);
while (!reader.EndOfStream)
    Console.WriteLine(reader.ReadLine());
reader.Close();
```

For writing, **`StreamWriter`** is your friend:

```
FileStream stream = File.Open("Settings.txt", FileMode.Create);
StreamWriter writer = new StreamWriter(stream);
writer.Write("IsFullScreen=");
writer.WriteLine(true);
writer.Close();
```

Note the file mode supplied as the second parameter on each of those **`File.Open`** calls. **`StreamWriter`**'s **`Write`** and **`WriteLine`** methods are almost like **`Console`**'s.

With this approach, our reading and writing do not need to happen all at once. We can read and write in small chunks over time, which is the main reason for using streams over the simpler **`WriteAllLines`** and **`ReadAllLines`**. Additionally, we can pass the **`StreamWriter`** or **`StreamReader`** (or just the raw stream) to other methods or objects. This ability lets you break complex serialization and deserialization in whatever way your design needs.

The **`BinaryReader`** and **`BinaryWriter`** classes are similar but use binary representations instead of text. Binary formats are typically much more compact but are also not easy for a human to open and read. For example, you could use **`writer.Write(1001)`**, which writes the **`int`** value **`1001`** into 4 bytes in binary, then use **`reader.ReadInt32()`**, which assumes the next four bytes are an **`int`** and decodes them as such.

Working with streams is far trickier than **`File.ReadAllText`**-type methods. For example, it is easy to accidentally leave a file open or close it too early. (Notably, all of these stream-related objects implement **`IDisposable`**, and should be disposed of when done, as described in Level 47.) I recommend using the simpler file methods when practical to avoid this complexity, especially if you are new to programming.

Find a Library

One big problem with everything we have talked about so far is writing all of the serialization and deserialization code. That can be tough to get right. Even something as simple as the CSV format has tricky corner cases. While you can always work through such details, finding somebody else's code that already solves the problem is often easier.

When possible, pick a widely used file format instead of inventing your own. With common file formats, it is easy to find existing code that does the serialization for you (or at least the heavy lifting). There are libraries—reusable packages of code—out there for standard formats like XML, JSON, and YAML. Using these libraries means you do not have to figure out all the details yourself. Level 48 has more information on libraries.

Before writing voluminous, complex serialization code, consider if an existing format and library can make your life easier.

Challenge The Long Game 100 XP

The island of Io has a long-running tradition that was destroyed when the Uncoded One arrived. The inhabitants of Io would compete over a long period of time to see who could press the most keys on the keyboard. But the Uncoded One's arrival destroyed the island's ability to use the Medallion of Files, and the historic competitions spanning days, weeks, and months have become impossible. As a True Programmer, you can use the Medallion of Files to bring back these long-running games to the island.

Objectives:

- When the program starts, ask the user to enter their name.
- By default, the player starts with a score of 0.
- Add 1 point to their score for every keypress they make.
- Display the player's updated score after each keypress.
- When the player presses the Enter key, end the game. **Hint:** the following code reads a keypress and checks whether it was the Enter key: `Console.ReadKey().Key == ConsoleKey.Enter`
- When the player presses Enter, save their score in a file. **Hint:** Consider saving this to a file named *[username].txt*. For this challenge, you can assume the user doesn't enter a name that would produce an illegal file name (such as `*`).
- When a user enters a name at the start, if they already have a previously saved score, start with that score instead.

Level 40

Pattern Matching

Speedrun

- Pattern matching categorizes data into one or more categories based on its type, properties, etc.
- Switch expressions, switch statements, and the `is` keyword all use pattern matching.
- Constant pattern: matches if the value equals some constant: `1` or `null`
- Discard pattern: matches anything: `_`
- Declaration pattern: matches based on type: `Snake s` or `Monster m`
- Property pattern: checks an object's properties: `Dragon { LifePhase: LifePhase.Ancient }`
- Relational patterns: `>= 3`, `< 100`
- `and`, `or`, and `not` patterns: `LifePhase.Ancient or LifePhase.Adult`, `not LifePhase.Wyrmling`
- `var` pattern: matches anything but also puts the result into a variable: `var x`
- Positional pattern: used for multiple elements, tuples, or things with a `Deconstruct` method to provide sub-patterns for each of the elements: `(Choice.Rock, Choice.Scissors)`
- Switches also have case guards, using the `when` keyword: `Snake s when s.Length > 2`

Programming is full of categorization problems, where you must decide which of several categories an object fits in based on its type, properties, etc.

- Is today a weekend or a weekday?
- In a game where you fight monsters, how many experience points should the player receive after defeating it?
- In the game of Rock-Paper-Scissors, given the player's choices, which player won?

You can solve these problems with the venerable `if` statement, but C# provides another tool designed specifically for these situations: *pattern matching*. Pattern matching lets you define categorization rules to determine which category an object fits in. You can use pattern matching in switch expressions, switch statements, and the `is` keyword.

The Constant Pattern and the Discard Pattern

We got our first glimpse of patterns in Level 10 when we made our pirate-themed menu:

```
string response = choice switch
{
    1 => "Ye rest and recover your health.",
    2 => "Raiding the port town get ye 50 gold doubloons.",
    3 => "The wind is at your back; the open horizon ahead.",
    4 => "'Tis but a baby Kraken, but still eats toy boats.",
    _ => "Apologies. I do not know that one."
};
```

This level was our introduction to patterns, though we didn't know it at the time.

In a switch expression, each arm is defined by a pattern on the left, followed by the **=>** operator, followed by the expression to evaluate if the pattern is a match (*pattern expression* => *evaluation expression*). Each pattern is a rule that determines if the object under consideration fits into the category or not. The switch expression above uses the two most basic patterns: the *constant pattern* and the *discard pattern*.

The first four lines show the constant pattern, which decides if there is a match based on whether the item exactly equals some constant value, like the literals **1**, **2**, **3**, or **4**.

The last switch arm uses the discard pattern: **_**. This pattern is a catch-all pattern, matching anything and everything. In C#, a single underscore usually represents a discard, signifying that what goes in that spot does not matter. Here, it indicates that there is nothing to check—that there are no constraints or rules for matching the pattern. Because it matches anything, when it shows up, it should always be the very last pattern.

But these two patterns are only the beginning.

The Monster Scoring Problem

Having a realistic and complex problem can help illustrate the different patterns we will be learning. In this level, we will use the following problem: in a game where the player fights monsters, given some **Monster** instance, determine how many points to award the player for defeating it. In code, we might write this problem like so:

```
int ScoreFor(Monster monster)
{
    // ...
}
```

Let's also define what a **Monster** is:

```
public abstract record Monster;
```

Monsters in a real game would likely have more than that, but it's all we need right now. Other monster types are derived from **Monster**:

```
public record Skeleton() : Monster;
```

A more complex subtype might add additional properties:

```
public record Snake(double Length) : Monster;
```

Anacondas are more challenging to defeat than mere garter snakes; the player deserves a larger reward for defeating them.

Here is a **Dragon** type that builds on two enumerations:

```
public record Dragon(DragonType Type, LifePhase LifePhase) : Monster;
public enum DragonType { Black, Green, Red, Blue, Gold }
public enum LifePhase { Wyrmling, Young, Adult, Ancient }
```

Each dragon has a type and a life phase. Different types of dragons and different life phases make for more formidable challenges worth more points.

And here is an orc with a sword that has properties of its own:

```
public record Orc(Sword Sword) : Monster;
public record Sword(SwordType Type);
public enum SwordType { WoodenStick, ArmingSword, Longsword }
```

The sword has a type: a longsword, an arming sword, or a wooden stick. It may be a stretch to call a **WoodenStick** a sword, but it is always worth compromising the design for stupid humor! (Please don't quote me on that.)

We could make more, but this is enough to make meaningful patterns.

The Type and Declaration Patterns

The *type pattern* matches anything of a specific type. For example, the following code uses the type pattern to look for the **Snake** and **Dragon** types:

```
int ScoreFor(Monster monster)
{
    return monster switch
    {
        Snake => 7,
        Dragon => 50,
        _  => 5
    };
}
```

Snake => 7 and **Dragon => 50** are both type patterns. (The last is another discard pattern.) If the monster is the named type, it will be a match. So this code will return 7 for snakes, 50 for dragons, and 5 otherwise. This pattern is a match even for derived types. A pattern like **Monster => 2** would match every kind of monster, regardless of its specific subtype.

The *declaration pattern* is similar but additionally gives you a variable that you can use in the body afterward. So we could change this so that longer snakes are worth more points:

```
int ScoreFor(Monster monster)
{
    return monster switch
    {
        Snake s => (int)(s.Length * 2),
        Dragon  => 50,
        _       => 5
    };
}
```

I also changed the whitespace to make all of the **=>** elements line up. This spacing is a common practice to increase the readability of the code. It puts it into a table-like format.

Case Guards

Switches have a feature called a *guard expression* or a *case guard*. These allow you to supply a second expression that must be evaluated before deciding if a specific arm matches. We can use this to have our snake rule apply only to long snakes:

```
int ScoreFor(Monster monster)
{
    return monster switch
    {
        Snake s when s.Length >= 3 => 7,
        Dragon                     => 50,
        _                          => 5
    };
}
```

A snake with a length of 4 will match both expressions on the first arm. A snake with a length of 2 only matches the pattern but not the guard, so the first arm will not be picked.

Once we have a guard expression, it can make sense to have multiple declaration patterns for the same type. The following gives 7 points for long snakes and 3 for others.

```
int ScoreFor(Monster monster)
{
    return monster switch
    {
        Snake s when s.Length >= 3 => 7,
        Snake                      => 3,
        Dragon                     => 50,
        _                          => 5
    };
}
```

Order matters. If you reverse the top two lines, the length-based pattern would never get a chance to match. If the compiler detects this, it will create a compiler error to flag it.

You can use case guards with any pattern.

The Property Pattern

The property pattern lets you define a pattern based on an object's properties. For example, ancient dragons should be worth far more than other life phases. We can show that with the pattern below:

```
int ScoreFor(Monster monster)
{
    return monster switch
    {
        Snake s when s.Length >= 3                 => 7,
        Dragon { LifePhase: LifePhase.Ancient } => 100,
        Dragon                                     => 50,
        _                                          => 5
    };
}
```

You can list multiple properties, separating them with commas:

```
Dragon { LifePhase: LifePhase.Ancient, Type: DragonType.Red } => 110,
```

The property pattern only matches if the type is correct, and each property is also a match.

You can also list a variable name for the matched object after the curly braces:

```
Dragon { LifePhase: LifePhase.Ancient } d => 100,
```

If you had a need, you could use the variable **`d`** in the expression after the **`=>`**.

If you don't want to demand a specific subtype, you can leave the type off in a property pattern: **`{ SomeProperty: SomeValue } => 2`**. The monster class has no properties, so it isn't helpful in this specific situation. But it is useful in other circumstances.

Nested Patterns

Some patterns allow you to use smaller sub-patterns within them. This is called a *nested pattern*. Each property in a property pattern is a nested pattern. The code above uses a nested constant pattern (**`LifePhase.Ancient`**), but we could have used any other pattern. To illustrate, here are some nested patterns for orcs with swords of different types:

```
Orc { Sword: { Type: SwordType.Longsword } }   => 15,
Orc { Sword: { Type: SwordType.ArmingSword } } => 8,
Orc { Sword: { Type: SwordType.WoodenStick } } => 2,
```

The highlighted code above is a property pattern inside another property pattern. But the inner pattern is not just limited to property patterns. It can be anything.

For nested property patterns, there's also a convenient shortcut:

```
Orc { Sword.Type: SwordType.Longsword }   => 15,
Orc { Sword.Type: SwordType.ArmingSword } => 8,
Orc { Sword.Type: SwordType.WoodenStick } => 2,
```

Nested patterns give you lots of flexibility but also begin to complicate code. It is important to be conscientious of the complexity of these patterns. You will inevitably be back and modify them again, and you will need to remember what they do.

Relational Patterns

We used a case guard for our snake pattern earlier, but an alternative would have been a relational pattern. These use **`>`**, **`<`**, **`>=`**, and **`<=`** to match a range of values. Now that we know the property pattern, we can replace our switch guard with the following:

```
Snake { Length: >= 3 } => 7,
```

The **`>= 3`** part is a relational pattern. It happens to be nested here, but we could use it at the top level without anything else if our switch were for an **`int`** instead of a **`Monster`**. **`>`**, **`<`**, and **`<=`** all work in the same way. We will see another example in a moment.

THE AND, OR, AND NOT PATTERNS

If you need a pattern that combines multiple sub-patterns, you can use **and** and **or**, which work like the **&&** and **||** operators do in Boolean logic. For example, if we want to give bonus points for dragons that are either ancient or adult, we use **or** in a nested property pattern:

```
Dragon { LifePhase: LifePhase.Adult or LifePhase.Ancient } => 100,
```

This saves us from needing to write out two entirely different switch arms.

Suppose we want to give short snakes (length under 2) 1 point, medium snakes (between 2 and 5) 3 points, and long snakes (longer than 5) 7 points. We could do this:

```
Snake { Length: < 2 }           => 1,
Snake { Length: >= 2 and <= 5 } => 3,
Snake { Length: > 5 }           => 7,
```

The **not** pattern negates the pattern after it. The following matches any non-wyrmling dragon:

```
Dragon { LifePhase: not LifePhase.Wyrmling } => 50,
```

THE POSITIONAL PATTERN

The *positional pattern* is useful when making decisions based on more than one value. Our monster scoring problem only involves a single object, so let's change to a different problem: deciding who won a game of Rock-Paper-Scissors.

Let's say we have the following two enumerations, one that represents a player's selection and one that represents the outcome of a game:

```
public enum Choice { Rock, Paper, Scissors }
public enum Player { None, One, Two }
```

We want to make a **DetermineWinner** method that tells us which player won when given the players' choices.

With the positional pattern, we can switch on multiple items:

```
Player DetermineWinner(Choice player1, Choice player2)
{
    return (player1, player2) switch
    {
        (Choice.Rock,     Choice.Scissors)      => Player.One,
        (Choice.Paper,    Choice.Rock)          => Player.One,
        (Choice.Scissors, Choice.Paper)         => Player.One,
        (Choice a,        Choice b) when a == b => Player.None,
        _                                       => Player.Two
    };
}
```

This code combines the two items at the switch's start, allowing patterns to use both elements.

All but the last of these is a positional pattern. It lists sub-patterns for each piece. For the overall pattern to match, each sub-pattern must match its corresponding part. Like with the property pattern, these sub-patterns can be any other pattern necessary. The above uses constant patterns in the first three lines and the declaration pattern in the fourth.

Deconstructors and the Positional Pattern

The positional pattern works when you lump two or more items together in parentheses. It also works on a single thing if it has a deconstructor (Level 34). The deconstructor will be used to extract the object's parts and attempt to match the pieces with the corresponding elements of the positional pattern.

If a type has a deconstructor, you can optionally prefix a type, suffix a variable name, or both:

```
Dragon (DragonType.Blue, LifePhase.Wyrmling) d => 100,
```

The var Pattern

The *var pattern* is somewhere between the declaration pattern (like **`Choice a`**) and the discard pattern. It does not check for a specific type, but does give you access to a new variable. Earlier, we did this:

```
(Choice a, Choice b) when a == b => Player.None,
```

We could have used the **`var`** pattern since the type **`Choice`** was already known:

```
(var a, var b) when a == b => Player.None,
```

The **`var`** pattern matches any type, but the variable it declares is useable in both the guard expression and the expression on the right of the **`=>`**.

Parenthesized Patterns

When your patterns start to get complex (especially when you use many **`and`** and **`or`** patterns), you can place parts of a pattern in parentheses to group things and enforce the order. The following is not very practical, but illustrates the point:

```
Snake { Length: (>2 and <5) or (>100 and <1000) } => 20,
```

Patterns with Switch Statements and the is Keyword

Switch expressions are the most common way to use patterns, but you can also use them in a switch statement and with the **`is`** keyword.

Switch Statements

Here is a version of **`DetermineWinner`** that uses a switch statement instead of a switch expression:

```
Player DetermineWinner(Choice player1Choice, Choice player2Choice)
{
    switch (player1Choice, player2Choice)
    {
        case (Choice.Rock, Choice.Scissors):
        case (Choice.Paper, Choice.Rock):
        case (Choice.Scissors, Choice.Paper):
            return Player.One;
        case (Choice a, Choice b) when a == b:
            return Player.None;
        default:
            return Player.Two;
    }
}
```

A switch expression that uses patterns is usually cleaner than its switch statement counterpart. But sometimes, the statement-based nature is needed or desirable.

Switch statements allow you to stack multiple patterns for a single arm, as shown above for the first three patterns. If you declare new variables while doing this (the **`var`** or declaration patterns), their names can sometimes cause conflicts with each other.

The `is` Keyword

Switches let you put an item into one of several rule-based categories. The **`is`** keyword enables you to check if something is in a single category or not. Here is a simple example:

```
void TellUserAboutMonster(Monster monster)
{
    Console.WriteLine("There's a monster!");
    if (monster is Snake)
        Console.WriteLine("Why did it have to be snakes?");
}
```

We are not just limited to the declaration pattern, though it is commonly combined with **`is`**. Any of the patterns are available to us.

The **`is`** keyword cannot use guard expressions, but we can always extend it with an **`&&`**.

Summary

The following table summarizes the different patterns available in C#:

Pattern Name	Description	Examples
constant pattern	matches a specific constant value	`3` or `null`
discard pattern	matches anything (a catch-all)	`_`
`var` pattern	matches anything and gives it a new name	`var x`
type pattern	matches if the object is the type listed at run time	`string`
declaration pattern	matches if the object is the type listed at run time and gives you a new variable	`string s`
property pattern	matches if the properties listed match the specified sub-patterns	`{ LifePhase: LifePhase.Wyrmling }`
relational pattern	matches if the object is `>`, `<`, `>=`, or `<=` the value provided	`>10, <= 1000`
`and` pattern	matches if both sub-patterns are a match	`>1 and <10`
`or` pattern	matches if either (or both) sub-patterns are a match	`<1 or >10`
`not` pattern	matches if the sub-pattern does not	`not null`
positional pattern	matches if each element in a tuple/deconstructor match their listed sub-patterns	`(Choice.Rock, Choice.Scissors)`

Challenge The Potion Masters of Pattren 150 XP

The island of Pattren is home to skilled potion masters in need of some help. Potions are mixed by adding one ingredient at a time until they produce a valuable potion type. The potion masters will give you the Patterned Medallion if you help them make a program to build potions according to the rules below:

- All potions start as water.
- Adding stardust to water turns it into an elixir.
- Adding snake venom to an elixir turns it into a poison potion.
- Adding dragon breath to an elixir turns it into a flying potion.
- Adding shadow glass to an elixir turns it into an invisibility potion.
- Adding an eyeshine gem to an elixir turns it into a night sight potion.
- Adding shadow glass to a night sight potion turns it into a cloudy brew.
- Adding an eyeshine gem to an invisibility potion turns it into a cloudy brew.
- Adding stardust to a cloudy brew turns it into a wraith potion.
- Anything else results in a ruined potion.

Objectives:

- Create enumerations for the potion and ingredient types.
- Tell the user what type of potion they currently have and what ingredient choices are available.
- Allow them to enter an ingredient choice. Use a pattern to turn the user's response into an ingredient.
- Change the current potion type according to the rules above using a pattern.
- Allow them to choose whether to complete the potion or continue before adding an ingredient. If the user decides to complete the potion, end the program.
- When the user creates a ruined potion, tell them and start over with water.

LEVEL 41

OPERATOR OVERLOADING

Speedrun

- Operator overloading lets you define how certain operators work for types you make: **+**, **-**, *****, **/**, **%**, **++**, **--**, **==**, **!=**, **>=**, **<=**, **>**, **<**. For example: **`public static Point operator +(Point p1, Point p2) => new Point(p1.X + p2.X, p1.Y + p2.Y);`**
- All operators must be **`public`** and **`static`**.
- Indexers let you define how the indexing operator works for your type with property-like syntax: **`public double this[int index] { get => items[index]; set => items[index] = value; }`**
- Custom conversions allow the system to cast to or from your type: **`public static implicit operator Point3(Point2 p) => new Point3(p.X, p.Y, 0);`**
- Custom conversions can be **`implicit`** or **`explicit`**. Use **`implicit`** when no data is lost; use **`explicit`** when data is lost.

The built-in types have some features that our types have been missing so far. For example, with **`int`**, you can do addition with the **`+`** operator:

```
int a = 2;
int b = 3;
int c = a + b;
```

With arrays, lists, and dictionaries, you can use the indexing operator:

```
int[] numbers = new int[] { 1, 2, 3 };
numbers[1] = 88;
Console.WriteLine(numbers[1]);
```

And you can cast from certain types to others. This code casts a **`char`** to a **`short`**:

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

You can define how operators, indexers, and casting conversions work for the types you create. That is the topic of this level.

Operator Overloading

We have encountered many different operators in our journey. You can define how some of these operators work for new types you make. Defining how these operators work is called *operator overloading*. For example, the **`string`** class has done this with **+** to allow things like **`"Hello " + "World!"`**.

You cannot overload all operators, but most work. For example, you can overload your typical math operators: **+**, **-**, *****, **/**, and **%**, the unary **+** and **-** (the positive and negative signs), as well as **++** and **--**. You can also overload the relational operators (**>**, **<**, **>=**, **<=**, **==**, and **!=**), but these must be done in matching pairs. If you overload **==** you must also overload **!=**, and same with **<** and **>**, or **>=** and **<=**. You cannot directly overload the compound assignment operators (**+=**, **-=**, etc.), but when you overload **+**, **+=** is automatically handled for you. You cannot overload the indexing operator (**[]**) as described in this section, but you can use an indexer as described in the next section instead.

You cannot invent new operators in C#. I'd like to create what I call the *marketing operator*, **<=>**, for those times when "you could save up to 15% or more by switching" (**`savings <=> 0.15`**). Alas, that is not possible. (But you can always make a method.)

Defining an Operator Overload

Before we overload an operator, we need a class where an operator makes sense. We will use the following **`Point`** record here, but you can overload operators on any class or struct.

```
public record Point(double X, double Y); // Could have also made a simple class.
```

The math world has a clear-cut definition for adding points together: you add each corresponding component together. Given the points at **`(2, 3)`** and **`(1, 8)`**, addition is done like so: **`(2+1, 3+8)`** or **`(3, 11)`**.

Defining this in code looks like this:

```
public record Point(double X, double Y)
{
    public static Point operator +(Point a, Point b) =>
                                         new Point(a.X + b.X, a.Y + b.Y);
}
```

Operators are essentially a special kind of static method. They must be marked both **`public`** and **`static`**, and you cannot define operators in unrelated types. At least one of the parameters must match the type you are putting the operator in.

What distinguishes an operator from a simple static method is the **`operator`** keyword and the operator's symbol instead of a name.

The above code uses an expression body, but it can also use a block body like any method.

With this operator defined, we can put it to use:

```
Point a = new Point(2, 3);
Point b = new Point(1, 8);

Point result = a + b;
Console.WriteLine($"({result.X}, {result.Y})");
```

Let's do a second example: scalar multiplication. Scalar multiplication is when we take a point and multiply it by a number. It has the effect of scaling the point by the amount indicated by the number. The point **`(1, 3)`** multiplied by **`3`** results in the point **`(1*3, 3*3)`** or **`(3, 9)`**.

```
public static Point operator *(Point p, double scalar) =>
                                  new Point(p.X * scalar, p.Y * scalar);
public static Point operator *(double scalar, Point p) => p * scalar;
```

I have defined two **`*`** operators rather than one. More on that in a second, but these two operators allow us to do this:

```
Point p = new Point(1, 3);
Point q = p * 3;
Point r = 3 * p;
```

When operators use different types, order often matters. If you want to support both orderings, you must define the operator twice, once for each order. One can call the other, so you don't have to copy and paste the code. If we left off the second definition, the class would support **`p * 3`** but not **`3 * p`**.

If you are defining one of the unary operators, your operator would have just a single parameter, such as this negation operator:

```
public static Point operator -(Point p) => new Point(-p.X, -p.Y);
```

When to Overload Operators

In any situation where you might overload an operator, you could also choose to use a method instead. How do you decide which to use? Use the version that makes your code the most understandable. The syntax around using operators is very concise (**`a - b`** is far shorter than **`a.Subtract(b)`**), but it only helps understandability if the meaning of subtraction for your type is intuitive.

INDEXERS

You can define how the indexing operator (**`[]`**) works with your class by making one or more *indexers*. These have some commonality with operators but have more in common with properties. In some ways, they are like a property with a parameter, and some people refer to them as *parameterful properties*. (That's a mouthful; I prefer *indexer*.)

Here's an example indexer in a simple **`Pair`** class:

```
public class Pair
{
    public int First { get; set; }
    public int Second { get; set; }

    public double this[int index]
    {
        get
        {
            if (index == 0) return First;
            else return Second;
        }
        set
        {
            if (index == 0) First = value;
            else Second = value;
        }
    }
}
```

You can see the similarities between this and a property. Both have getters and setters, and both have that implicit **`value`** parameter in the setter, etc. The only real difference is that you also have access to the parameters defined in the square brackets—your index variables. The **`number`** variable is accessible in both the getter and the setter.

An indexer need not be limited to just **`int`**s. The following lets you use **`'a'`** and **`'b'`**:

```
public double this[char letter]
{
    get
    {
        if (letter == 'a') return First;
        else return Second;
    }
    set
    {
        if (name == 'a') First = value;
        else Second = value;
    }
}
```

In this case, I'd generally recommend just using the **`First`** and **`Second`** properties, but an indexer makes a lot of sense when the allowed indices are large or not known ahead of time.

An indexer can also have multiple parameters. The following indexer lets you access items in a 2D grid of numbers (a matrix):

```
public int this[int row, int column]
{
    // ...
}
```

A type can define as many indexers as needed, as long as they have different parameters.

Index Initializer Syntax

Any type that defines an indexer can take advantage of *index initializer syntax*. Like object initializer syntax, you can use this to set up an object through its indexers:

```
Pair p = new Pair()
{
    [0] = 1,
    [1] = -4
};
```

The above code is virtually the same as this:

```
Pair p = new Pair();
p[0] = 1;
p[1] = -4;
```

Perhaps a better illustration of this syntax is the **`Dictionary`** class. The code below uses index initializer syntax to set up a dictionary of colors based on their name:

```
Dictionary<string, Color> namedColors = new Dictionary<string, Color>
{
    ["red"]    = new Color(1.0, 0.0, 0.0),
    ["orange"] = new Color(1.0, 0.64, 0.0),
    ["yellow"] = new Color(1.0, 1.0, 0.0)
};
```

Custom Conversions

In C#, you can cast between types that don't have a direct inheritance relationship. For example:

```
int a = (int)3.0; // Explicit cast or conversion from a double to an int.
double b = 3;     // Implicit cast or conversion from an int to a double.
```

You can define custom conversions for the types you create. Custom conversions are done much like operator overloading but with some differences. To illustrate, let's rename our **`Point`** class from earlier to **`Point2`**, and then let's also say we have a **`Point3`** with an **`X`**, **`Y`**, and **`Z`** property, for representing a 3D location.

```
public record Point2(double X, double Y);
public record Point3(double X, double Y, double Z);
```

Converting between these two types might be nice. You could even think of a **`Point2`** as a **`Point3`** with a **`Z`** coordinate of **`0`**.

We must consider what data may be lost in conversions of this sort. Going from **`Point2`** to **`Point3`** loses nothing. **`Point3`** can carry the **`X`** and **`Y`** values over and use **`0`** as the default **`Z`** value. But going from **`Point3`** to **`Point2`** will lose the **`Z`** component. It is likely reasonable for conversion from **`Point2`** to **`Point3`** to happen automatically. It is likely *un*reasonable for conversion from **`Point3`** to **`Point2`** to happen automatically. We see the same thing with **`int`** and **`long`**. Casting from an **`int`** to a **`long`** happens implicitly, but going the other way requires explicitly stating the cast:

```
int a = 0;
long b = a;     // Implicit cast.
int c = (int)b; // Explicit cast.
```

A **`long`** can accurately store every possible value that **`int`** can hold, so the conversion is safe. An **`int`** cannot contain all possible values of a **`long`**, so the conversion has risk and must be written out. When defining conversions for **`Point2`** and **`Point3`**, we must keep this in mind.

Here is our first conversion, from a **`Point2`** to a **`Point3`**:

```
public static implicit operator Point3(Point2 p) => new Point3(p.X, p.Y, 0);
```

A custom conversion is much like an operator (indeed, it is defining the typecasting operator). The two main differences are the **`implicit`** keyword and the name **`Point3`**. Each operator will be either **`implicit`** or **`explicit`**. This choice indicates whether the cast can happen automatically (**`implicit`**) or must be spelled out (**`explicit`**). You list the type to convert to in the position where a name would go. The above is a conversion from **`Point2`** (based on the parameter type) to **`Point3`** (based on the "name").

The body performs the conversion. Like any method, you can use an expression body or a block body.

Custom conversions must be defined in one of the types involved in the conversion, so this operator must go into either **`Point2`** or **`Point3`**.

With this conversion added to either **`Point2`** or **`Point3`**, we can now write code like this:

```
Point2 a = new Point2(1, 2);
Point3 b = a;
```

Even with no inheritance relationship between the two, the conversion from **`Point2`** to **`Point3`** will happen automatically. The compiler will see the need for a conversion, look for an appropriate one, and apply it.

The conversion from **`Point3`** to **`Point2`** loses data, so we define that as an **`explicit`** conversion:

```
public static explicit operator Point2(Point3 p) => new Point2(p.X, p.Y);
```

We chose **`explicit`** instead of **`implicit`** because we do not want somebody to lose data without specifically asking for it.

```
Point3 a = new Point3(1, 2, 3);
Point2 b = (Point2)a;
```

The Pitfalls of Custom Conversions and Some Alternatives

Custom conversions create new objects, which can have unexpected consequences for reference types. Suppose we make **`Point3`**'s properties settable and also make this method:

```
void MoveLeft(Point3 p) => p.X--;
```

Consider this usage:

```
Point2 point = new Point2(0, 0);
MoveLeft(point);
```

This code seems reasonable at first glance. **`point`** is converted to a **`Point3`** before **`MoveLeft`** is called, and then the point is shifted. However, the conversion to a **`Point3`** creates a new object, and it is this new object that is passed to **`MoveLeft`**. The original **`Point2`** is unchanged.

Errors like this are hard to notice. Some recommend avoiding custom conversions entirely because of subtle issues like this. When and how to use custom conversions is your choice, but knowing the alternative is useful. We could make this simple method instead:

```
public Point3 ToPoint3() => new Point3(X, Y, 0);
```

This requires us to call **`ToPoint3()`**, which is far more likely to raise a red flag:

```
Point2 point = new Point2(0, 0);
MoveLeft(point.ToPoint3());
```

It is more apparent that you are passing a separate object with this code.

You could also define a constructor that does the conversion:

```
public Point3(Point2 p) : this(p.X, p.Y, 0) { }
```

The conversion also stands out more clearly:

```
Point2 point = new Point2(0, 0);
MoveLeft(new Point3(point));
```

Custom conversions are not evil, but keep this consequence in mind as you write them.

Knowledge Check — Operators — 25 XP

Check your knowledge with the following questions:

1. **True/False**. Operator overloading allows you to define a new operator such as **`@@`**.
2. **True/False**. You can overload all C# operators.
3. **True/False**. Operator overloads must be **`public`**.

Answers: (1) False. **(2)** False. **(3)** True.

Challenge — Navigating Operand City — 100 XP

The City of Operand is a carefully planned city, organized into city blocks, lined up north to south and east to west. Blocks are referred to by their coordinates in the city, as we saw in the Cavern of Objects. The inhabitants of the town use the following three types as they work with the city's blocks:

```
public record BlockCoordinate(int Row, int Column);
public record BlockOffset(int RowOffset, int ColumnOffset);
public enum Direction { North, East, South, West }
```

`BlockCoordinate` refers to a specific block's location, **`BlockOffset`** is for relative distances between blocks, and **`Direction`** specifies directions. As we saw with the Cavern of Objects, rows start at 0 at the north end of the city and get bigger as you go south, while columns start at 0 on the west end of the city and get bigger as you go east.

The city has used these three types for a long time, but the problem is that they do not play nice with each other. The town is the steward of *three* Medallions of Code. They will give each of them to you if you can use them to help make life more manageable. Use the code above as a starting point for what you build.

In exchange for the Medallion of Operators, they ask you to make it easy to add a **`BlockCoordinate`** with a **`Direction`** and also with a **`BlockOffset`** to get new **`BlockCoordinates`**. Add operators to **`BlockCoordinate`** to achieve this.

Objectives:

- Use the code above as a starting point.
- Add an addition (**+**) operator to **`BlockCoordinate`** that takes a **`BlockCoordinate`** and a **`BlockOffset`** as arguments and produces a new **`BlockCoordinate`** that refers to the one you would arrive at by starting at the original coordinate and moving by the offset. That is, if we started at **`(4, 3)`** and had an offset of **`(2, 0)`**, we should end up at **`(6, 3)`**.
- Add another addition (**+**) operator to **`BlockCoordinate`** that takes a **`BlockCoordinate`** and a **`Direction`** as arguments and produces a new **`BlockCoordinate`** that is a block in the direction indicated. If we started at **`(4, 3)`** and went east, we should end up at **`(4, 4)`**.
- Write code to ensure that both operators work correctly.

Challenge Indexing Operand City 75 XP

In exchange for the Medallion of Indexers, the city asks for the ability to index a **`BlockCoordinate`** by a number: **`block[0]`** for the block's row and **`block[1]`** for the block's column. Help them in this quest by adding a get-only indexer to the **`BlockCoordinate`** class.

Objectives:

- Add a get-only indexer to **`BlockCoordinate`** to access items by an index: index 0 is the row, and index 1 is the column.
- **Answer this question:** Does an indexer provide many benefits over just referring to the **`Row`** and **`Column`** properties in this case? Explain your thinking.

Challenge Converting Directions to Offsets 50 XP

Operanders often use both the **`Direction`** and the **`BlockOffset`** in casual communication: "go north" or "go two blocks west and one block south." However, it would be convenient to convert a direction to a **`BlockOffset`**. For example, the direction north would become an offset of **`(-1, 0)`**. Operanders offer you their final medallion, the Medallion of Conversions, if you can add a custom conversion in **`BlockOffset`** that converts a **`Direction`** to a **`BlockOffset`**.

Objectives:

- Add a custom conversion to **`BlockOffset`** that converts from **`Direction`** to **`BlockOffset`**.
- **Answer this question:** This challenge didn't call out whether to make the conversion explicit or implicit. Which did you choose, and why?

LEVEL 42

QUERY EXPRESSIONS

Speedrun

- Query expressions are a special type of statement that allows you to extract specific pieces from a data collection and return it in a particular format or organization.
- Query expressions are made of multiple clauses.
- **`from`** identifies the collection that is being queried.
- **`select`** identifies the data to be produced.
- **`where`** filters out elements in the query.
- **`orderby`** sorts the results.
- **`join`** combines multiple collections.
- **`let`** allows you to give a name to a part of a query for later reference.
- **`into`** continues queries where it would otherwise have terminated.
- **`group`** categorizes data into groups.
- All queries can be done using query syntax or with method calls.

Most programs deal with collections of data and need to search the data. This type of task is called a *query*. Here are some examples:

- In real estate, find all houses under $400,000 with 2+ bathrooms and 3+ bedrooms.
- In a project management tool, find all active tasks assigned to each person on the team.
- In a video game, find all objects close enough to an explosion to take splash damage.

C# has a type of expression designed to make queries easy. These expressions are *Language Integrated Queries* (*LINQ*) or simply *query expressions*. These query expressions are most commonly done on objects in memory—arrays, lists, dictionaries, etc. But LINQ also makes it possible for a LINQ query to retrieve data from an actual database such as MySQL, Oracle, or Microsoft SQL Server. The first is known as LINQ for Objects, and the second is LINQ for SQL. We will focus on querying objects in memory since it is the most versatile.

Anything we do with query expressions could have been done with **`if`** statements and loops. But as we will see, query expressions are often more readable and shorter.

Queries and `IEnumerable<T>`

Query expressions work on anything that implements **`IEnumerable<T>`**. That is virtually all collection types in .NET, including lists, arrays, and dictionaries. In this level, when I refer to collections, datasets, or sets, I'm referring to anything that implements **`IEnumerable<T>`**.

The logic for doing query expressions with **`IEnumerable<T>`** does not live in **`IEnumerable<T>`** itself. Instead, a set of extension methods (Level 34) implement the query expression functionality. There are almost 200 of these extension methods, so it is a good thing you do not have to implement all of them to define a new **`IEnumerable<T>`**!

The **`System.Linq.Enumerable`** class defines these extension methods. There is an implicit **`using`** directive for **`System.Linq`** in .NET 6+ projects, but if you are using an older version, you will need to add **`using System.Linq;`** to your files manually.

Sample Classes

We will use the following three classes in the samples in this level. You might find similar classes in a game. The **`GameObject`** class is the base class of potentially many types of objects found in the game, and **`Ship`** is one of those types. The **`Player`** class represents a game player, and **`GameObject`** instances are each owned by a player.

```
public class GameObject
{
    public int ID { get; set; }
    public double X { get; set; }
    public double Y { get; set; }
    public int MaxHP { get; set; }
    public int HP { get; set; }
    public int PlayerID { get; set; }
}

public class Ship : GameObject { }

public record Player(int ID, string UserName, string TeamColor);
```

If you are following along at home, you might also find the following setup code helpful:

```
List<GameObject> objects = new List<GameObject>();
objects.Add(new Ship { ID = 1, X=0, Y=0, HP = 50, MaxHP = 100, PlayerID = 1 });
objects.Add(new Ship { ID = 2, X=4, Y=2, HP = 75, MaxHP = 100, PlayerID = 1 });
objects.Add(new Ship { ID = 3, X=9, Y=3, HP = 0,  MaxHP = 100, PlayerID = 2 });

List<Player> players = new List<Player>();
players.Add(new Player(1, "Player 1", "Red"));
players.Add(new Player(2, "Player 2", "Blue"));
```

Query Expression Basics

You form a query expression out of a series of smaller elements called *clauses*. Each clause is like a station in an assembly line. It receives elements from the clause before it and supplies elements to the clause after it. Add new clauses as needed to get the result you want.

Query expressions begin with a *from* clause and end with a *select* clause. A *from* clause identifies the source of the data. A *select* clause indicates which part of the data to produce as a final result. The simplest query expression possible is this:

```
IEnumerable<GameObject> everything = from o in objects
                                     select o;
```

Above, I have put each clause on a separate line and used whitespace to line them up. That is not necessary, but it is a common practice. It makes it easier to understand.

Despite what I have done in most of this book, I will use **`var`** instead of spelling out the variable's type in most code samples in this level. Books don't have much horizontal space, and the long name detracts from the focus. But note that the result is an **`IEnumerable<GameObject>`**, not a **`List<GameObject>`**. Query expressions produce **`IEnumerable<T>`**.

The *from* clause is the first line: **`from o in objects`**. A *from* clause begins a query expression by naming the source of the query: **`objects`**. It also introduces a variable named **`o`**. A variable in a *from* clause is called a *range variable*. The rest of the query expression can use this variable. While more descriptive names are often better, query expressions are so small that C# programmers often use just a single letter.

The *select* clause is the second line: **`select o`**. A *select* clause starts with the **`select`** keyword, followed by an expression that computes the query expression's final result objects. The expression **`o`** is the simplest possible expression, taking **`o`** whole and unchanged. We will see more complex ones soon.

The result is an **`IEnumerable<GameObject>`** containing the exact same items as **`objects`**.

Let's try something more meaningful. This query expression grabs each object's **`ID`** instead of the entire object:

```
var ids = from o in objects
          select o.ID;
```

The result is an **`IEnumerable<int>`**, rather than **`IEnumerable<GameObject>`** because the *select* clause's expression produced an **`int`**. But the sky is the limit in what you can put in a *select* clause's expression. For example:

```
var healthText = from o in objects
                 select $"{o.HP}/{o.MaxHP}";
```

This query will give you a string for each object in the game with text like **`"0/50"`** or **`"92/100"`**.

How about this one?

```
var healthStatus = from o in objects
                   select (o, $"{o.HP}/{o.MaxHP}"); // Tuple
```

This query creates a tuple combining the original object with its health text. The type of **`healthStatus`** is **`IEnumerable<(GameObject, string)>`**. Query expressions make it easy to build weird, complex types for short-term use.

Filtering

A *where* clause provides an expression used to filter the elements passing by it in the assembly line. It includes an expression that must be true for the item to remain past the *where* clause. The following expression produces only game objects with non-zero hit points remaining:

```
var aliveObjects = from o in objects
                   where o.HP > 0
                   select o;
```

This expression can be any **bool** expression. You can make it as complex as you need.

While query expressions begin with a **from** and end with a **select**, the middle is far more flexible. The following applies multiple *where* clauses back to back:

```
var aliveObjects = from o in objects
                   where o.PlayerID == 4
                   where o.HP > 0
                   select o;
```

Ordering

An *orderby* clause will order items. This code will create an **IEnumerable<GameObject>** where the first item has the lowest **MaxHP**, then the next lowest, etc.

```
var weakestObjects = from o in objects
                     orderby o.MaxHP
                     select o;
```

You can reverse the order by placing the **descending** keyword at the end:

```
var strongestObjects = from o in objects
                       orderby o.MaxHP descending
                       select o;
```

The **ascending** keyword can also be used there, but that is the default, so there is usually no need.

If you need to break a tie, you can list multiple expressions to sort on, separated by commas:

```
var weakestObjects = from o in objects
                     orderby o.MaxHP, o.HP
                     select o;
```

This sorts by **MaxHP** primarily but resolves ties by looking at **HP**. You can name as many sorting criteria as you need with more commas.

You can use these middle-of-the-pipeline clauses however they are needed, in any order and number. For example:

```
var player4WeakestObjects = from o in objects
                            where o.PlayerID == 4
                            orderby o.HP
                            where o.HP > 0
                            orderby o.MaxHP
                            select o;
```

Few queries end up so complicated. There is rarely a need for it, and many programmers will split apart long queries to keep things clear. But keep in mind that ordering does make a difference in the results produced and also in speed.

Method Call Syntax

If you don't like all of these new keywords, you're in luck. You can write every query expression with method calls instead of keywords. (The compiler transforms the keywords into method calls anyway.) This approach is called *method call syntax*. Instead of the **where** keyword, you use the **`Where`** method. Instead of the **`select`** keyword, you use the **`Select`** method. Consider this keyword-based query:

```
var results = from o in objects
              where o.HP > 0
              orderby o.HP
              select o.HP / o.MaxHP;
```

With method call syntax, it would look like this:

```
var results = objects
                   .Where(o => o.HP > 0)
                   .OrderBy(o => o.HP)
                   .Select(o => o.HP / o.MaxHP);
```

These methods typically have delegate parameters, so lambda expressions are common.

The conversion from keywords to method calls should not always be literal. For example, while a keyword-based expression must end with a **`select`**, even if that is just **`select o`**, method call syntax does not require ending with a **`Select`**. You do not need to do **`Select(o => o)`**.

Some people prefer the conciseness of the keyword-based version. Others feel like method calls are just more natural. Yet others will use some of both, depending on which seems cleaner for the specific query. You can decide for yourself which you like better.

Unique Methods

Method call syntax can do everything the keywords can do, plus a few things for which there are no keywords. Here are a few of the most useful.

`Count` allows you to either count the total items in the collection or the number that meet some specific condition:

```
int totalItems = objects.Count();
int player1ObjectCount = objects.Count(x => x.PlayerID == 1);
```

`Any` and **`All`** can tell you if any or every element in the collection meets some condition:

```
bool anyAlive = objects.Any(y => y.HP > 0);
bool allDead = objects.All(y => y.HP == 0);
```

`Skip` lets you skip a few items at the beginning, while **`Take`** lets you grab the first few while dropping the rest:

```
var allButFirst = objects.OrderBy(m => m.HP).Skip(1);
var firstThree = objects.OrderBy(m => m.HP).Take(3);
```

`Average`, **`Sum`**, **`Min`**, and **`Max`** let you do math with the items or with some aspect of the item:

```
int longestName          = players.Max(p => p.UserName.Length);
int shortestName         = players.Min(p => p.UserName.Length);
double averageNameLength = players.Average(p => p.UserName.Length);
int totalHP              = objects.Sum(o => o.HP);
```

There are many more, and when you want to explore them, you can use Visual Studio's IntelliSense to dig around and see what's out there (Bonus Level A).

Advanced Queries

SIDE QUEST

Few query expressions need more than the above, but there is quite a bit more to query expressions when you need to get fancy. This section covers more advanced usages of the clauses we already saw and looks at a few additional clause types.

Let's start by fleshing out one more detail of the **from** clause. If you are confident that everything in a collection is of some specific derived type, you can name the derived type instead, making it the type used in subsequent clauses. The code below assumes all game objects are the **Ship** class, and the result is an **IEnumerable<Ship>** instead of an **IEnumerable<GameObject>**:

```
IEnumerable<Ship> ships = from Ship s in objects
                          select s;
```

If you are wrong, it will throw an **InvalidCastException**, so you must know ahead of time or filter it to just that type first.

The method call syntax equivalent of this is **gameObjects.Cast<Ship>()** if you know they are all ships or **gameObjects.OfType<Ship>()** if you are unsure and want to filter down to only those of that type.

Multiple `from` Clauses

Query expressions start with a **from** clause, but you can have many if you want to work with multiple collections at once. Two **from** clauses will allow us to look at each pairing of items. If **GameObject** had a **CollidingWith(GameObject)** method, we could write the following code to get a collection of all pairings that are colliding (intersecting) objects:

```
var intersections = from o1 in objects
                    from o2 in objects
                    where o1 != o2          // Don't compare an object to itself.
                    where o1.CollidingWith(o2)
                    select (o1, o2);
```

Multiple **from** clauses can quickly cause performance issues. If we have 10 game objects, we will evaluate 10×10 or 100 total comparisons. If we have 1000 game objects, it will be 1000×1000 or 1,000,000 total comparisons.

Joining Multiple Collections Together

In other situations, you want to combine two collections through a common link. Rather than looking at every item in one collection paired with every item in a second, we want to see only pairings that match each other. In the database world, this is called a *join*. In our example classes, **GameObject** has a **PlayerID** property corresponding to **Player**'s **ID** property. We can determine which color a game object should be by finding the player associated with the game object and using that player's color using a *join* clause:

```
var objectColors = from o in objects
                   join p in players on o.PlayerID equals p.ID
                   select (o, p.TeamColor);
```

The *join* clause introduces the second collection and a second range variable, **p**. After the **on**, you can specify which part of each range variable to use in determining a pairing. The order matters. The first collection's range variable must come before the equals; the second collection's range variable must go after. You typically refer to a property from each variable for comparison, but more complex expressions are allowed if necessary.

A *join* clause produces all successful pairings. If an object in one collection had no match, it would not appear in the results. If an object pairs with several items in the other collection, each pairing appears. However, in many situations, this is avoided by other parts of the software. For example, if two players had the same ID, then we would see a pairing of an object with both players from a *join* clause. But we would typically ensure each player's ID is unique.

Once past the join, you can use both range variables in your clauses, knowing that the two belong together.

The `let` Clause

A **`let`** clause defines another variable in the middle of a query expression that you can use afterward. This clause is great if you need to use some computation repeatedly:

```
var statuses = from o in objects
               let percentHealth = o.HP / o.MaxHP * 100
               select $"{o.ID} is at {percentHealth}%.";
```

Continuation Clauses

A *select* clause typically ends a query expression, but you can keep it going with an *into* clause (also called a *continuation clause*). This clause introduces a new range variable and begins a new query expression on the tail of the previous one:

```
var deadStrongObjectIDs = from o in objects
                          where o.MaxHP > 50
                          select (o.ID, o.HP, o.MaxHP, o.HP / o.MaxHP)
                          into objectHealth
                          where objectHealth.HP == 0
                          select objectHealth.ID;
```

The original range variable is not accessible past the *into*. Essentially, a new query expression has started. Some people will adjust whitespace to make this stand out visually:

```
var deadStrongObjectIDs = from o in objects
                          where o.MaxHP > 50
                          select (o.ID, o.HP, o.MaxHP, o.HP / o.MaxHP)
                into objectHealth
                          where objectHealth.HP == 0
                          select objectHealth.ID;
```

Alternatively, you can also just write it as two separate statements:

```
var strongObjects = from o in objects
                    where o.MaxHP > 50
                    select (o.ID, o.HP, o.MaxHP, o.HP / o.MaxHP);
var deadObjectIDs = from s in strongObjects
                    where s.HP == 0
                    select s.ID;
```

Grouping

A *group by* clause puts the items into groups. It is a second way to end a query expression. The following will group all of our objects by their owning player:

```
IEnumerable<IGrouping<int, GameObject>> groups = from o in objects
                                                 group o by o.PlayerID;
```

Notice the return type. The result is a collection of groupings. The **`IGrouping<Tkey, TElement>`** interface extends **`IEnumerable<TElement>`** augmented with a shared key. Here, the key is the player ID, and the items in the collection will be all of the objects that belong to the player.

A *group by* clause contains two expressions. The first (**`o`** in the code above) determines the final elements of each group. The second (**`o.PlayerID`** in the code above) determines what the shared key is for each group. Either can be as complex as needed.

Group Joins

The final clause type is the *group join*, combining elements of both grouping and joining. These can be very elaborate clauses, formed of many pieces that can each be complex. A situation that might call for a group join is if you wanted each player and their owned objects. A simple grouping is not sufficient because a player with no game objects does not end up with a group at all. A simple join is not enough because it doesn't do grouping.

A group join starts the same as a simple join, then includes an **`into`**:

```
var playerObjects = from p in players
                    join o in objects on p.ID equals o.PlayerID into ownedObjects
                    select (Player: p, Objects: ownedObjects);

foreach (var playerObjectSet in playerObjects)
{
    Console.WriteLine($"{playerObjectSet.Player.UserName} has the following:");
    foreach (var gameObject in playerObjectSet.Objects)
        Console.WriteLine(gameObject.ID);
}
```

All items from the second collection associated with the object from the first are bundled into a new **`IEnumerable<T>`** and given a new name (**`ownedObjects`**). You get a result even if that is an empty collection.

The above code combines the player with its objects in a tuple and displays the results.

Even simple group joins are often complicated; try to keep them understandable as you build them.

Deferred Execution

Arrays and lists store their data in memory. In contrast, query expressions do not need to compute all results immediately. A query expression is almost like defining the machinery or assembly line for producing items without actually creating them. Instead, the results are built a little at a time, only as the next item is needed. This approach is called *deferred execution*.

The upshot of deferred execution is that it is gentle on memory. If you have a vast set of items to dig through, they do not all need to be put in an array to use them. You can look at them one at a time. And if you figure out what you need after only a few items, the rest of them never need to be computed and placed in memory at all.

On the other hand, if you need to go through all items repeatedly, you end up computing the collection's contents more than once, which wastes time. If you are in this situation, use the **`ToArray`** and **`ToList`** methods, which will materialize the entire set into an array or list:

```
var aliveObjects = from o in objects
    where o.HP > 0
    select o;

List<GameObject> aliveObjectsList = aliveObjects.ToList();

foreach(var aliveObject in aliveObjectsList)
    Console.WriteLine($"{aliveObject.ID} is alive!");

foreach(var aliveObject in aliveObjectsList)
    aliveObject.HP--;
```

The cost for computing the collection happens once (in the **`ToList()`** method), and the iteration over the collection in both **`foreach`** loops stays fast.

In general, you should prefer deferred execution when you can. Only materialize the entire collection into a list or array when processing the whole set more than once.

Not all query expressions can pull off deferred execution. For example, the **`Count`** method must walk through every element to compute the answer. In situations like these, immediate evaluation will happen out of necessity.

LINQ TO SQL

Our focus in this level has been on using query expressions on collections in memory. This scheme is called *LINQ to Objects*. But query expressions can also work against data in a database. This scheme is called *LINQ to SQL*. Unfortunately, this complex subject demands knowledge of databases beyond what this book can cover.

However, the syntax is identical, and the best part is that your query (or at least parts of it) will run inside the database engine itself. Thus, your C# code is automatically translated to the database's query language and runs over there. (This is where the name “Language INtegrated Query” comes from.)

LINQ to SQL isn't a wholesale replacement for interacting with a database. For example, you cannot write data in a query expression. But it makes specific database tasks far easier.

Knowledge Check — Queries — 25 XP

Check your knowledge with the following questions:

1. What clause (keyword) starts a query expression?
2. What clause filters data?
3. **True/False**. You can order by multiple criteria in a single **`orderby`** clause.
4. What clause combines two related sets of data?

Answers: (1) **`from`** clause. **(2)** **`where`** clause. **(3)** True. **(4)** **`join`** clause.

Challenge — The Three Lenses — 100 XP

The Guardian of the Medallion of Queries, Lennik, has long awaited when he can return the Medallion to a worthy programmer. But he only wants to give it to somebody who truly understands the value of queries. He requires you to build a solution to a simple problem three times over. Lennik gives you the following array of positive numbers: [1, 9, 2, 8, 3, 7, 4, 6, 5]. He asks you to make a new collection from this data where:

- All the odd numbers are filtered out, and only the even should be considered.
- The numbers are in order.
- The numbers are doubled.

For example, with the array above, the odd/even filter should result in 2, 8, 4, 6. The ordering step should result in 2, 4, 6, 8. The doubling step should result in 4, 8, 12, 16 as the final answer.

Objectives:

- Write a method that will take an `int[]` as input and produce an `IEnumerable<int>` (it could be a list or array if you want) that meets all three of the conditions above *using only procedural code*—`if` statements, switches, and loops. **Hint:** the static `Array.Sort` method might be a useful tool here.
- Write a method that will take an `int[]` as input and produce an `IEnumerable<int>` that meets the three above conditions using a *keyword-based query expression* (`from x`, `where x`, `select x`, etc.).
- Write a method that will take an `int[]` as input and produce an `IEnumerable<int>` that meets the three above conditions using a *method-call-based query expression*. (`x.Select(n => n + 1)`, `x.Where(n => n < 0)`, etc.)
- Run all three methods and display the results to ensure they all produce good answers.
- **Answer this question:** Compare the size and understandability of these three approaches. Do any stand out as being particularly good or particularly bad?
- **Answer this question:** Of the three approaches, which is your personal favorite, and why?

LEVEL 43

THREADS

Speedrun

- Creating threads allows your program to do more than one thing at a time: **`Thread thread = new Thread(MethodNameHere);  thread.Start();`**
- If you need to pass something to a thread, your start method must have a single **`object`** parameter, which is supplied with **`thread.Start(theObject);`**
- Wait for a thread to finish with **`Thread.Join`**.
- Threads that share data can cause problems. If you do this, protect critical sections with a lock: **`lock (aPrivateObject) { /* code in here is thread safe */ }`**.

In the beginning, all computers had only one processor, allowing them to do just one thing at a time. Modern computers typically have multiple processors, letting them do many things simultaneously. More specifically, they usually have multiple cores on the same processor chip. Four cores and eight cores are commonplace, and 16 or 32 are not uncommon either.

You can leverage this power and get long-running jobs done significantly faster if you write your code correctly. The concept of running multiple things at the same time is called *concurrency*. The next two levels cover two of the primary flavors of concurrency. We will use multiple "threads" of execution to do multi-threaded programming in this level.

THE BASICS OF THREADS

A *thread* is an independent execution path in a program. Every program has at least one thread in it. When we run a typical C# program, a thread is created and begins running our main method, one instruction at a time.

A program can create additional threads, and each can run its own code. When a program does this, it becomes a *multi-threaded* application. Each thread gets its own stack to manage its method calls, but all threads in your program share the same heap, letting them share data.

Modern computers have many processors, but they also usually juggle many applications and services, each with one or more threads. There are typically far more threads than there are processors. The operating system has a *scheduler* that decides when to let each thread have a chance to run. Like a juggler, the scheduler ensures each thread gets frequent opportunities to run on a processor without letting any languish. The scheduler's job is complicated, weighing factors like each thread's priority in the system and how long it has been waiting for a turn.

When the scheduler decides to swap out one thread for another, it takes time. It must remember the thread's state so it can be restored later and then unpack the replacement's previous state so that it can resume. This swap is called a *context switch*. This swap is unproductive time, but if context switches don't happen often enough, threads starve, and their work doesn't get done.

Your program has little control over the scheduler. You will not know when your threads will get a chance to run, nor is it obvious which code will execute first if it is happening on different threads. That makes multi-threaded programming far more complicated than single-threaded programming. It is sometimes worth the trouble, and sometimes not.

We'll cover the key elements of multi-threaded programming, but this is a complex issue that we cannot fully cover in this book.

USING THREADS

Before creating more threads, we must first identify work that can run independently. This is one of the most complex parts of multi-threaded programming. It is an art and a science, and it takes patience and practice to get good at it.

You want work that is entirely (or almost entirely) independent of the rest of your code, and that will take a while to run. If it is intertwined with everything else, it does not make sense to run it separately. If it is too small in size, it won't be worth the overhead of creating a whole other thread. Threads are comparatively expensive to make and maintain.

Let's keep it simple and do multi-threading with the simple task of counting to 100:

```
void CountTo100()
{
    for (int index = 0; index < 100; index++)
        Console.WriteLine(index + 1);
}
```

The **`System.Threading.Thread`** class captures the concept of a thread. The **`System.Threading`** namespace is automatically added in .NET 6+ projects, but if you target an older version of .NET, you will want to add a **`using System.Threading;`** to the top of the file.

When you create a new thread, you give it the method to run in its constructor (Level 36).

```
Thread thread = new Thread(CountTo100);
```

In this case, the method must have a **`void`** return type and no parameters.

Once created, you start the thread by calling its **`Start()`** method:

```
thread.Start();
```

After calling **`Start()`**, the new thread will begin running the code in the method you supplied, while your program's original "main" thread will continue to the next statement below **`thread.Start();`**. Both threads will run in parallel. The scheduler will let each thread run for a while, juggling them and the threads in other processes.

A complete multi-threaded program may look like this:

```
Thread thread = new Thread(CountTo100);
thread.Start();
Console.WriteLine("Main Thread Done");

void CountTo100()
{
    for (int index = 0; index < 100; index++)
        Console.WriteLine(index + 1);
}
```

Both threads write stuff in the console window, but you cannot predict how the scheduler will run them. The text “Main Thread Done” could appear before all the numbers, after all the numbers, or somewhere in the middle. I just ran it once and got the following:

```
1
2
3
Main Thread Done
4
...
```

Rerunning it produces a different order.

One thread can wait for another to finish before proceeding using **`Thread`**’s **`Join`** method. For example, the following makes two threads that count to 100 and waits for both to finish:

```
Thread thread1 = new Thread(CountTo100);
thread1.Start();
Thread thread2 = new Thread(CountTo100);
thread2.Start();

thread1.Join();
thread2.Join();
Console.WriteLine("Main Thread Done");
```

Repeatedly running this code and viewing its output can be extremely helpful for understanding how threads are scheduled. It’s not just a back-and-forth. Each thread gets a chunk of time in (seemingly) unpredictable lengths. One thread will display the first 13 numbers, and then the second gets to 22, then the first gets up to 18, and so on.

You will not see “Main Thread Done” in the middle of the numbers this time because the main thread will not reach that line until both counting threads finish.

You cannot directly task a thread with additional methods to run, but you could design a system where tasks are placed in a list somewhere, and the thread runs indefinitely, checking to see if new jobs have appeared for it to run. Once a thread finishes, it is over. You would just make a new thread for any other work.

Sharing Data with a Thread

Our first pass with threads did not allow them to share any data. The method the thread ran had no parameters and a **`void`** return type. Alternatively, we can use a single **`object`**-typed parameter and pass in an object that allows the main thread and the new thread to share information. This object is supplied when calling the thread's **`Start`** method:

```
MultiplicationProblem problem = new MultiplicationProblem { A = 2, B = 3 };
Thread thread = new Thread(Multiply);
thread.Start(problem);

thread.Join();

Console.WriteLine(problem.Result);

void Multiply(object? obj)
{
    if (obj == null) return; // Nothing to do if it is null.
    MultiplicationProblem problem = (MultiplicationProblem)obj;
    problem.Result = problem.A * problem.B;
}

class MultiplicationProblem
{
    public double A { get; set; }
    public double B { get; set; }
    public double Result { get; set; }
}
```

The parameter's type must be **`object`**. You will have to downcast to the right type in the new thread's method.

This shared object can have properties for all the inputs and results the thread may need, allowing the data to be shared with the original thread.

There are other ways that multiple threads can share data. The new thread has access to any accessible static methods and fields. If the thread is running an instance method, it will also have access to that object's instance data. That leads to this pattern:

```
Operation operation = new Operation(1, "Hello");
Thread thread = new Thread(operation.Run);
thread.Start();

public class Operation
{
    public int Number { get; }
    public string Word { get; }

    public Operation(int number, string word) { Number = number; Word = word; }

    public void Run() { /* Insert long task using Number and Word. */ }
}
```

There is a distinct danger to sharing data among threads, as we will soon see.

Sleeping

The static **`Thread.Sleep`** method pauses a thread for a bit.

```
Thread.Sleep(1000); // 1 second
```

The **Sleep** method is static, and it makes the current thread pause. The sleep time is in milliseconds (1000 milliseconds is one second). When you do this, the thread will give up the rest of its currently scheduled time and won't get another chance until after the time specified.

THREAD SAFETY

Any time two threads share data, there is a danger of them simultaneously modifying the data in ways that hurt each other. If the shared data is immutable (read-only), this problem solves itself. Consider even just this simple example of two threads that both increment a **_number** field that they both have access to:

```
SharedData sharedData = new SharedData();
Thread thread = new Thread(sharedData.Increment);
thread.Start();

sharedData.Increment();

thread.Join();

Console.WriteLine(sharedData.Number);

class SharedData
{
    private int _number;
    public int Number => _number;
    public void Increment() => _number++;
}
```

The main thread and the new thread do nothing but call the **Increment** method, which adds one to the variable. This program seems innocent enough, and you would expect when the program finishes, the output will be **2**. But consider how this could go wrong. **_number++;** is the same as **_number = _number + 1;**. It retrieves the value out of **_number**, adds one to it, then stores the updated value back in **_number**. It is a three-step process, and due to the scheduling nature of threads, we cannot guarantee when each thread will run any given step in that process. This won't usually cause problems, but the following scenario is possible:

1. Thread 1 reads the current value out of **_number** (a value of 0).
2. Thread 1 computes the new value (1).
3. Thread 2 reads the current value out of **_number** (still 0!).
4. Thread 2 computes the new value (1).
5. Thread 2 updates **_number** (1).
6. Thread 1 updates **_number** (1 again!).

Even though two threads went through the logic of incrementing the variable, we got an unexpected outcome. Programmers call this type of problem variously a *threading issue*, a *concurrency issue*, or a *synchronization issue*. These are some of the most frustrating problems in programming. They may work 99.999% of the time, and the logic seems to be fine at a glance. But once in a blue moon, our timing is unlucky, and things break in subtle ways. These concurrency issues can be incredibly tough to spot and fix—a reason to avoid unnecessary multi-threaded programming.

While there are tools that address concurrency issues (which we'll discuss in a moment), they open up the possibility of other problems that can be just as painful.

When code does not use shared data, only uses immutable (read-only) shared data, or correctly handles its access to shared data, it is considered *thread-safe*. Not everything needs to be thread-safe, but you will want to make it so if multiple threads use it.

Locks

The first step in addressing concurrency issues is identifying the code that must be made thread-safe. These are usually places where we need an entire section of code to run to completion once it begins, as seen from the outside world.

In the sample above, that is the line **`_number++;`**. Either thread can run that statement to completion first, but once a thread starts working with that variable, it must be allowed to finish before another thread begins.

These sections are called *critical sections*. Only one thread at a time should be able to enter a critical section. Identifying critical sections is half the battle.

Once you have identified a critical section, it is time to protect it. This protection is done with *mutual exclusion*—a fancy way of saying whichever thread gets there first gets to keep going, and everybody else must wait for their turn. It is very much like going into a public restroom and locking the door behind you to prevent others from coming in while you're using it. (Every good book needs at least one potty analogy, right?)

C# has many options for enforcing mutual exclusion, but C#'s **`lock`** keyword is the main one. Things that enforce mutual exclusion are called a *mutex*. A lock is a type of mutex. It is easier to show how to use a **`lock`** statement than to describe it. The code below illustrates protecting our **`_number`** variable with a **`lock`** statement:

```
SharedData sharedData = new SharedData();
Thread thread = new Thread(sharedData.Increment);
thread.Start();

sharedData.Increment();

thread.Join();

Console.WriteLine(sharedData.Number);

class SharedData
{
    private readonly object _numberLock = new object();

    private int _number;

    public int Number
    {
        get
        {
            lock (_numberLock)
            {
                return _number;
            }
        }
    }

    public void Increment()
    {
        lock (_numberLock)
        {
            _number++;
        }
    }
}
```

Wrap the critical sections inside of a **`lock`** statement. Lock statements are associated with a specific object. The first part of the **`lock`** statement, **`lock (_numberLock)`**, is referred to as *acquiring the lock*. No thread can proceed past this step until it acquires the lock for the object. While one thread has the lock, others are temporarily barred from entry. When a thread reaches the end of the **`lock`** statement, the lock is released, and another thread can acquire it.

You can use any reference-typed object in a lock, but creating a new plain **`object`** instance is commonplace. It is one of the few places where a simple **`object`** instance is practical. However, you want to avoid locking on objects that are not private.

You don't want to make a lock object too broadly or too narrowly used. Generally, you will make a single lock object to protect both read and write access to a single piece of data or group of related data elements. The above code had two lock statements that used the same lock object. If we added a **`Decrement`** method, we would reuse the same object. If we had other data in this class that was modified independently, we'd use a separate lock object for it.

Threads can acquire multiple locks if needed, but you should avoid these situations when you can. Imagine needing to use both the keyboard and mouse to do a job, and I grab the keyboard, and you grab the mouse. You're waiting for me to release the keyboard while I'm waiting for you to release the mouse. We both spend the rest of our lives waiting for the other item to become available. This is called a *deadlock* and is one of many concurrency-related issues.

Multi-threaded programming is trickier than single-threaded programming. Avoid it when you can, but when you can't, apply the tools and techniques here to make it work. And plan on a little extra time to hunt down these hard-to-find bugs.

Challenge The Repeating Stream 150 XP

In Threadia, there is a stream that generates numbers once a second. The numbers are randomly generated, between 0 and 9. Occasionally, the stream generates the same number more than once in a row. A repeat number like this is significant—an omen of good things to come. Unfortunately, since the Uncoded One's arrival, Threadians haven't been able to monitor the stream while it produces numbers. Either the stream generates numbers while nobody watches, or they watch while the stream produces no numbers. The Threadians offer you the Medallion of Threads willingly and ask you to use it to make both possible at the same time. Build a program to generate numbers while simultaneously allowing a user to flag repeats.

Objectives:

- Make a **`RecentNumbers`** class that holds at least the two most recent numbers.
- Make a method that loops forever, generating random numbers from 0 to 9 once a second. **Hint: `Thread.Sleep`** can help you wait.
- Write the numbers to the console window, put the generated numbers in a **`RecentNumbers`** object, and update it as new numbers are generated.
- Make a thread that runs the above method.

- Wait for the user to push a key in a second loop (on the main thread or another new thread). When the user presses a key, check if the last two numbers are the same. If they are, tell the user that they correctly identified the repeat. If they are not, indicate that they got it wrong.
- Use **`lock`** statements to ensure that only one thread accesses the shared data at a time.

LEVEL 44

ASYNCHRONOUS PROGRAMMING

Speedrun

- Asynchronous programming lets tasks run in the background, scheduling continuations or callbacks to happen with the asynchronous task results when it completes.
- The `Task` and `Task<TResult>` classes can be used to schedule tasks to run asynchronously: `Task.Run(() => { ... });`
- You can write code to run after the task completes by awaiting the task: `await someTask;`
- You can only use the `await` keyword in methods that have the `async` keyword applied to them: `async Task<int> DoStuff() { ... }`

Another model of concurrent programming is *asynchronous programming*. In this model, you begin a long-running request and perform other work instead of waiting for it to complete. When the job finishes, you are notified and can continue onward with its results. The opposite is called *synchronous programming*, and it is what we have done in the rest of this book.

You use this asynchronous model in your daily life:

- You text a friend to see if they want to meet for lunch. You don't stare at the screen waiting for a response, but go on with your day. When your friend responds, you get a notification on your phone and can plan the rest of your day.
- You order at a fast-food restaurant, then sit and talk with your family while it is prepared. When it is ready, your name or number is called out, and you get your food and eat it.
- When you apply for a job, you update your resume and submit your application, and then it is in the business's hands. You go back to your life and wait for a response. In the acting world, getting called back in for a second interview or audition is given the name "callback." That's a word we'll use in a programming context later in this level.

It is also helpful in various programming situations:

- You want to pull down leaderboard data, user data, or even a software update from a server. A network request takes time, and you don't want the program to hang while awaiting the response.
- You're saving off a small mountain of data to a file.
- You have a complex, long-running computation behind the scenes.

In short, a long-running task needs to happen, but we want to be notified when it finishes instead of stopping all work while waiting.

A Sample Problem

We'll use the following problem to illustrate the key points in this level. Suppose we want to run a computation at a space base on Jupiter's moon Europa. It takes time to transmit through space, so we don't want to hold up other work while this happens. Here is some code that represents this task, done synchronously:

```
int result = AddOnEuropa(2, 3);
Console.WriteLine(result);

int AddOnEuropa(int a, int b)
{
    Thread.Sleep(3000); // Simulate light delay. It should be far longer!
    return a + b;
}
```

This program is time-consuming but has one thing going for it: it is easy to understand. Keep this simplicity in mind as we explore various asynchronous solutions below.

Threads and Callbacks

Before we get to the best solution, let's consider a couple of solutions we could do with the knowledge we already have. These other solutions help us understand the final solution.

We could put this work on a separate thread, as we saw in the previous level. The following code uses several advanced C# features, but it is about as concise as we can get with threads:

```
int result = 0;
Thread thread = new Thread(() => result = AddOnEuropa(2, 3));

thread.Start();

// Do other work here

thread.Join();

Console.WriteLine(result);
```

This code uses delegates, lambdas, and closures (covered in an optional Side Quest section).

Even though I have gone to great lengths to simplify this code, it is still far uglier than the synchronous version. Plus, this still has a problem: we don't know when it gets done! That comment line is hiding stuff. If it hides less than three seconds of work, we'll still be waiting at the **`Join`**. If it does more than three seconds of work, it will delay showing the results.

Another approach would be to give the long-running operation a delegate to invoke when it completes. A delegate like this is known as a *callback*:

```
AddOnEuropa(2, 3, result => Console.WriteLine(result));

void AddOnEuropa(int a, int b, Action<int> callback)
{
    Thread thread = new Thread(() =>
    {
        Thread.Sleep(3000);
        int result = a + b;
        callback(result);
    });
    thread.Start();
}
```

Once the slow work completes, the thread invokes the delegate to finish the job. At this point, the main thread can continue to other tasks, knowing that the callback will run when the time is right. This code is comparatively difficult to read.

USING TASKS

C# has a concept called a *task* representing a job that can run in the background. Some tasks produce a result of some sort, while others do not, similar to how a typical method can have a **void** return type or return a specific value. Some other languages have a similar concept but call it a *promise*. That is a good name for it because it captures the idea of a task well: "I don't have an **int** for you yet, but I promise I'll have one when I finish."

C# uses two classes for representing asynchronous tasks: **Task** and **Task<T>**. Use **Task** for tasks that produce no specific result (like a **void** method) and the generic **Task<T>** for tasks that promise an actual result. Both of these are in the **System.Threading.Tasks** namespace, which is one of the namespaces automatically included for you in new projects. If you're working in older projects, you may need to add a **using** directive (Level 33) for that namespace.

Task and Task<T> Basics

Let's begin our exploration with the basics of the **Task** and **Task<T>** classes. Our long-running **AddOnEuropa** method can return a **Task<int>**—a promise of an **int** in the future—instead of a plain **int**:

```
Task<int> AddOnEuropa(int a, int b) { /* ??? */ }
```

Let's look at how **AddOnEuropa** can make a task. Perhaps the simplest version is to create a new task that does not even run asynchronously—just a finished task with a specific value:

```
Task<int> AddOnEuropa(int a, int b)
{
    Thread.Sleep(3000);
    int result = a + b;
    return Task.FromResult(result);
}
```

This version has not achieved much. It will all run synchronously on the calling thread and produce a finished **Task** object at the end. But creating new, finished tasks has its place.

We want this to run in the background asynchronously, so let's do this instead:

```
Task<int> AddOnEuropa(int a, int b)
{
    Task<int> task = new Task<int>(() =>
    {
        Thread.Sleep(3000);
        return a + b;
    });
    task.Start();
    return task;
}
```

This version creates a **Task<int>** object, supplying a delegate (Level 36) for the task to run. The code above uses a lambda statement (Level 38) to define the task's work. The task doesn't begin executing this code until you call its **Start()** method.

It is easy to forget to call **Start()**, so the alternative below is usually better:

```
Task<int> AddOnEuropa(int a, int b)
{
    return Task.Run(() =>
    {
        Thread.Sleep(3000);
        return a + b;
    });
}
```

The static **Task.Run** method handles creating a task and starting it all at once. That makes our code simpler. **Task.Run** is the preferred way to begin new tasks for most situations.

We will revisit this method and make it even better later, but let's turn our attention to the calling side. How do you interact with a **Task** or **Task<T>** object returned by a method?

The first thing you can do with a task (**Task** or **Task<T>**) is call its **Wait** method. This suspends the current thread and waits until the task finishes:

```
Task<int> additionTask = AddOnEuropa(2, 3);
additionTask.Wait();
```

For tasks of the generic **Task<T>** variety, you will probably want the computed result, accessible through the **Result** property:

```
Task<int> additionTask = AddOnEuropa(2, 3);
additionTask.Wait();
int result = additionTask.Result;
Console.WriteLine(result);
```

If the task is still running, **Result** will automatically wait for the task to finish, so calling both **Wait()** and **Result** is redundant.

Calling **Wait** or **Result** is philosophically the same thing as calling **Thread.Join**. While we are using tasks, we have not received any substantial asynchronous benefits yet.

One improvement we can make is to create a second task as a *continuation* of the first. A continuation is essentially the same as a callback, just done with tasks. This code takes the **Console.WriteLine** statement and puts it into a continuation:

```
Task<int> additionTask = AddOnEuropa(2, 3);
Task addAndDisplay = additionTask.ContinueWith(t => Console.WriteLine(t.Result));
```

ContinueWith takes a delegate parameter with the type **Action<Task>**, allowing the continuation to inspect the results of the task before it. **ContinueWith** returns a second task that won't begin until the previous task finishes. There is also a generic overload of **ContinueWith** for when you want that continuation task to return a value itself:

```
Task<double> moreMath = additionTask.ContinueWith<double>(t => t.Result * 2);
```

There are a lot of other overloads for **ContinueWith**. We will soon be using a different approach for tasks, so we'll skip the details, but they are worth checking out someday.

The async and await Keywords

While the previous code is a decent way to work with tasks, the C# language has some built-in mechanisms that make working with tasks more straightforward: the **async** and **await** keywords. The **await** keyword is a convenient way to indicate that a method should asynchronously wait for a task to finish and schedule the rest of the method as a continuation.

```
int result = await AddOnEuropa(2, 3);
Console.WriteLine(result);

Task<int> AddOnEuropa(int a, int b)
{
    return Task.Run(() =>
    {
        Thread.Sleep(3000);
        return a + b;
    });
}
```

This code hides one crucial element: you can only use an **await** in a method marked with **async**. This code is in our main method, and the compiler automatically puts **async** on the generated method. But in every other method, you'll need to add that yourself:

```
async Task DoWork()
{
    int result = await AddOnEuropa(2, 3);
    Console.WriteLine(result);
}
```

This version of asynchronous code is much cleaner than the other versions we have seen. Our main method is the same except for the **await**. **AddOnEuropa** is also very similar to the original synchronous version, aside from the **Task.Run** and returning a **Task<int>** instead of a plain **int**. The compiler takes care of everything else for you. The compiler is even smart enough to propagate any exceptions thrown in the task, allowing the awaiting method to handle them (Level 35) despite potentially occurring on a separate thread.

One interesting thing about that **DoWork** method is that even though it claims to return a **Task**, there is no **return** statement. Similarly, if we had a method that claimed to return a **Task<int>**, we might see it return an **int**, but not a **Task<int>**. This is part of the compiler's magic to make this work correctly. The compiler generates code that returns a task; it is just invisible, hidden behind the **await**.

Not every method can be an **async** method. Only certain return types are supported. The following three are the most common by far: **void**, **Task**, or **Task<T>**.

Use **Task<T>** when you expect an asynchronous task to produce a result. Use **Task** when there is no specific result, but you still need to know when the task is done so you can perform work afterward. Use **void** only for "fire and forget" tasks where nobody will ever need to know when it finishes. If a method's return type is **void**, no other code will be able to await it.

An **async** method can have many **await**s in it. Consider the following two examples:

```
int result1 = await AddOnEuropa(2, 3);
int result2 = await AddOnEuropa(4, 5);
int result3 = await AddOnEuropa(result1, result2);
Console.WriteLine(result3);
```

And:

```
Task<int> firstAdd = AddOnEuropa(2, 3);
Task<int> secondAdd = AddOnEuropa(4, 5);

int result = await AddOnEuropa(await firstAdd, await secondAdd);
Console.WriteLine(result);
```

Both generally do the same thing (add on Europa three times) and use multiple **await**s. The location of the **await**s is important. In the first example, the second call to **AddOnEuropa** doesn't occur until after the first one completes. In the second example, the second call to **AddOnEuropa** happens before we await the first task. Those two long-running additions coincide.

The **async** and **await** keywords cause a lot of compiler magic to happen that makes your life easier. This is the approach to take when doing asynchronous programming. You will still find times to use things like **ContinueWith**, **Wait**, and **Result**, but most of the time, **async** and **await** are your best bet.

Who Runs My Code?

Tasks represent small asynchronous jobs; they are not threads and cannot directly run code. A thread must run every line of code.

Let's first talk about that **Task.Run** method, which is the typical entry into asynchronous code. When you call this method, it hands the task to the *thread pool*. As we learned in the previous level, threads are expensive to create and maintain. The thread pool is a collection of threads on standby, waiting to run small jobs like those represented by tasks. The thread pool itself (represented by the static **System.Threading.ThreadPool** class) manages when to make more threads or cleanup underutilized threads. It does an excellent job, so you rarely have to worry about tweaking its behavior. When a task is given to the thread pool, the next available thread will run the task's code. (Note that there are ways you can use **Task.Run** that make it run on a dedicated thread if a task is especially long-running. Few are that way.)

While the threads from the thread pool are often involved in asynchronous code, let's take a more detailed look. Consider this code:

```
async Task AsynchronousMethod()
{
    Console.WriteLine("A");
    Task task = Task.Run(() => { Console.WriteLine("B"); });
    Console.WriteLine("C");
    await task;
    Console.WriteLine("D");
}
```

Which thread will run each of the **Console.WriteLine**s?

The following is important because it is often misunderstood: just because a method returns a **Task** or has the **async** keyword does not make the whole thing run asynchronously! The original thread that called **AsynchronousMethod** will do as much work as it possibly can.

That initial thread will be the one to run **`Console.WriteLine("A");`**. It will also be the one to call **`Task.Run`**, which schedules **`"B"`** on the thread pool for later. If **`task`** is finished by the time **`await task;`** is called, the calling thread will continue through to the end of the method and also run the **`"D"`** line.

This code creates a task that will probably take some time to run, so most likely, the task will not be done before **`await task;`** is executed. In contrast, a task created with **`FromResult`** is complete the moment it is created. That reinforces the idea that just because a task is involved does not mean anything is happening asynchronously.

`Task.Run` schedules the **`Console.WriteLine("B");`** line on the thread pool. A thread from the thread pool will run it.

The **`await task;`** line does not require a thread to wait for it specifically. That is part of the **`async`** and **`await`** mechanisms the compiler generates, and it uses this point as a splitting point for continuations after the initial task.

So if a task must be awaited, who runs the continuation after the **`await`**? This question is the most complicated one to answer because it depends. In some contexts or situations, we want a specific thread (or set of threads) to run the continuation. For example, most UI frameworks are single-threaded—only one thread ("The UI Thread") can interact with controls on the screen. In these cases, the continuation needs to find its way back to this UI thread.

Tasks include the concept of a *synchronization context*. The job of a synchronization context is to represent one of these situations or contexts where a task originated, to allow the continuation to find its way home. When there is a synchronization context, the default behavior is to ensure the continuation runs there. For a UI application, this will put the continuation back on the UI thread. If there is no synchronization context, which is the case in a console application, then continuations will typically stay where they are at, and a thread pool thread will pick up **`Console.WriteLine("D");`**.

Most of the time, it doesn't matter what context a continuation runs in. Rather than having the system try to figure out how to get it back to the original context, it is sometimes helpful to tell the task to keep running any continuations on the thread pool using **`ConfigureAwait(false)`**:

```
await task.ConfigureAwait(false);
```

The **`false`** indicates that it should not return to the initial context, though **`true`** is the default.

One final point about who runs async code: some code does not need *any* thread to run it. For example, if you make an Internet request (or to Europa, as we pretended with our earlier example), we don't need a thread to sit there and use up CPU cycles waiting for it. The request can happen entirely off of our computer! In these cases, the asynchronous stuff isn't happening on *any* thread, which leaves all of our threads free to do other work.

Here is an example. We used **`Thread.Sleep(3000);`** earlier in this level to delay for a bit. There is another option: **`Task.Delay(3000)`**. This returns a **`Task`** that won't finish until the specified timeframe (in milliseconds) passes. That is, while **`Thread.Sleep`** puts a thread out of commission for a while, **`Task.Delay(3000)`** does not:

```
Task<int> AddOnEuropa(int a, int b)
{
    return Task.Run(async () =>
    {
        await Task.Delay(3000);
        return a + b;
    });
}
```

Note also the **`await`** keyword on the lambda. It is a little awkward just hanging out there, but we do need the **`async`** to use **`await`**, and this is how you do that with a lambda.

Some Additional Details

A few other asynchronous programming details are worth a brief discussion.

Exceptions

Exceptions (Level 35) bubble up the call stack from where they are thrown. The call stack is associated with a thread, and each thread has its own call stack. Tasks complicate this because the logic can bounce around among threads. The C# language designers wanted tasks to have a good exception handling experience, so they put a lot of work into this.

Instead of letting an exception escape a task directly, any exception that escapes a task's code is caught by the thread running it. It puts the task into an error state and stores the exception in the task. These exceptions are then rethrown on the **`await`** line when a task is awaited. This allows the awaiting code to handle those exceptions:

```
try
{
    await SlowOperation();
}
catch (InvalidOperationException)
{
    Console.WriteLine("I'm sorry, Dave, I'm afraid I can't do that.");
}
```

What happens if a task throws an exception but is never awaited? When the task is garbage collected, it will throw on a garbage collection thread and bring down your program. Because it happens later—sometimes much later—it can be tough to determine what led to the exception. You typically want to await all tasks that you run and handle any exceptions thrown.

Cancellation

Tasks support cancellation for long-running tasks. For tasks that you might want to cancel, you share a cancellation token with it. Anybody with access to the cancellation token can request that the task be canceled. But making the request does not cancel it automatically. The task's code must periodically check to see if a request has been made and then run any logic to cancel the operation.

The details of task cancellation are beyond what we can reasonably get into here, but it is helpful to know that the system facilitates it.

Awaitables

`Task` and **`Task<T>`** are the most common types used for asynchronous programming. But there are others. You can even define your own. Anything that you can apply the **`await`** keyword to is called an *awaitable*.

`ValueTask` and `ValueTask<T>` are analogous to `Task` and `Task<T>` but are value types instead of reference types (Level 14). These are far less common but useful if you're worried about memory usage and also typically know the result without running asynchronously.

`IAsyncEnumerable<T>` lets you build and process a collection a little at a time as results become available. (There is even special syntax when using this in a `foreach` loop.)

These aren't the only options, but between these, `Task`, and `Task<T>`, they are the most common.

Limitations

`async` and `await` lead to complex code behind the scenes. There are some limitations to combining them with certain other C# features. For example, you can't `await` something in a `lock` statement. These limitations are nuanced, so rather than writing them all out here, just know that they exist, and the compiler will point out any trouble spots.

More Information

Asynchronous programming is a tricky area of C#. We've covered the basics, but there is plenty more to learn. If you plan to do a lot with `async` and `await`, I recommend getting either or both of the following short books:

- *Async in C# 5.0*, by Alex Davies.
- *Concurrency in C# Cookbook, Second Edition,* by Stephen Cleary.

Knowledge Check **Async** **25 XP**

Check your knowledge with the following questions:

1. What keyword indicates that a method might schedule some of its work to run asynchronously?
2. What keyword indicates that code beyond that point should run once the task has finished?
3. Name three return types that can be used with the `async` keyword.
4. **True/False**. Code is always faster when run asynchronously.
5. What return type would be best for an `async` method that does the following:

   a. The work does not produce a value, but you need to know when it finishes.

   b. You do not care when the task completes.

   c. The task creates a result that you need to use afterward.

Answers: (1) `async`. **(2)** `await`. **(3)** `void`, `Task`, `Task<TResult>`, etc. **(4)** False. **(5)** a: `Task`. b: `void`. c: `Task<T>`.

Challenge **Asynchronous Random Words** **150 XP**

On the Island of Tasken, you meet Awat, who tells you that being a True Programmer can't be all that hard. His ancestors have been the stewards of the Asynchronous Medallion, yet Awat uses it as a food dish for his cat. "A thousand monoids with a thousand random generators will also eventually produce 'hello world'!" he claims. Indeed, they could, but you know it would take a while. With tasks, you can allow a human to pick a word and randomly generate the word asynchronously. Doing this will show Awat how long it will take to randomly generate the words "hello" and "world," convincing him that a Programmer's skills mean something.

Objectives:

- Make the method **`int RandomlyRecreate(string word)`**. It should take the string's length and generate an equal number of random characters. It is okay to assume all words only use lowercase letters. One way to randomly generate a lowercase letter is **`(char)('a' + random.Next(26))`**. This method should loop until it randomly generates the target word, counting the required attempts. The return value is the number of attempts.
- Make the method **`Task<int> RandomlyRecreateAsync(string word)`** that schedules the above method to run asynchronously (**`Task.Run`** is one option).
- Have your main method ask the user for a word. Run the **`RandomlyRecreateAsync`** method and await its result and display it. **Note:** Be careful about long words! For me, a five-letter word took several seconds, and my math indicates that a 10-letter word may take nearly two years.
- Use **`DateTime.Now`** before and after the async task runs to measure the wall clock time it took. Display the time elapsed (Level 32).

Challenge Many Random Words 50 XP

Awat is impressed with what you did in the last challenge but thinks it could be better. "Why not generate 'hello' and 'world' in parallel?" he asks. "You do that, and I'll let you take this medallion off of me."

Objectives:

- Modify your program from the previous challenge to allow the main thread to keep waiting for the user to enter more words. For every new word entered, create and run a task to compute the attempt count and the time elapsed and display the result, but then let that run asynchronously while you wait for the next word. You can generate many words in parallel this way. **Hint:** Moving the elapsed time and output logic to another **`async`** method may make this easier.

LEVEL 45

DYNAMIC OBJECTS

Speedrun

- The `dynamic` type instructs the compiler not to check a variable's type. It is checked while running instead. This is useful for dynamic objects whose members are not known at compile time.
- Avoid dynamic objects, except when they provide a clear, substantial benefit.
- `ExpandoObject` is best for simple, expandable objects.
- Deriving from `DynamicObject` allows for greater control in constructing dynamic members.

Types are a big deal in C#. We spend a lot of time designing new classes and structs to get precisely the right effect. We worry about inheritance hierarchies, carefully cast between types, and fret over parameter and return types.

In C#, types are considered "static" (not to be confused with the `static` keyword), meaning types don't change as the program runs. You cannot add new methods to a class or object, but that means the compiler can make strong guarantees that objects will truly have the methods and other members you invoke. This fact makes C# a *statically typed* language, or you could say that the compiler does *static type checking*.

The primary advantage of this is that the compiler can guarantee that everything you do is safe. If you call a method on an object, the compiler makes sure that the method exists. Any failures of this nature are caught by the compiler before your program even starts.

There are two variations in the opposite camp. First, we can have *dynamic type checking*. With dynamic type checking, variables (including parameters and return types) have no fixed type associated with them. The compiler cannot ensure any given member will exist. In exchange, there is usually less ceremony and formality around types. The second variation is *dynamic objects*. With dynamic objects, the objects themselves have no formal structure, sometimes being defined at creation time. Other times, they even allow methods and properties to be added and removed as the program runs.

C# can support both dynamic type checking and dynamic objects. But a word of caution is in order: these can be a useful tool, but the overwhelming majority of your C# code should be statically typed. Keep dynamic typing to a minimum, and only in circumstances where the benefits are clear and significant.

Dynamic Type Checking

With C#'s standard static type checking, the compiler can look at code and ensure that only objects or values of the right type are placed in a variable and that it only uses members that the type has. For example, in the code below, the compiler can ensure that **`text`** is only assigned **`string`** values, verify that **`string`** has a **`Length`** property, and check that **`Console`** has a **`WriteLine`** method with a single **`int`** parameter:

```
string text = "Hello, World!";
Console.WriteLine(text.Length);
```

You can have C# perform dynamic type checking for a variable by using the **`dynamic`** type:

```
dynamic text = "Hello, World!";
Console.WriteLine(text.Length);
```

Any variable can use this type. It tells the compiler to skip static type checking and instead make those checks while the program is running. The sample below abuses this, attempting operations that we know the **`string`** object won't have:

```
dynamic text = "Hello, World!";
text += Math.PI;
text *= 13;
Console.WriteLine(text.PurpleMonkeyDishwasher);
```

Each of these fails as the program is running with a **`RuntimeBinderException`**. On the other hand, this will work:

```
dynamic mystery = "Hello, World!";
Console.WriteLine(mystery.Length);
mystery = 2;
mystery += 13;
mystery *= 13;
Console.WriteLine(mystery);
```

The contents of **`mystery`** change from a **`string`** to an **`int`**! All of the operations we attempt are legitimate for whichever object **`mystery`** contains when the operation is used.

Behind the scenes, **`mystery`** becomes an **`object`**. The compiler will record metadata about method calls and use that metadata as the program is running to look up the correct member. Remember that if you treat a value type as an **`object`**, it is boxed (Level 28). That has an impact on how you use **`dynamic`** with value types.

Dynamic Objects

Dynamic objects are objects whose structure is determined while the program is running. In some cases, these dynamic objects can even change structure over the object's lifetime, adding and removing members. This is not what C# was designed for, but C# supports it for situations where this model can produce much simpler code. This primarily happens when using things made in a dynamic programming language or dynamic data formats.

Dynamic objects are built by implementing the **`IDynamicMetaObjectProvider`** interface. Implementing this interface tells the runtime how to look up properties, methods, and other members dynamically. But **`IDynamicMetaObjectProvider`** is extremely low-level and is both tedious and error-prone. Fortunately, there are some other tools you can use in most situations instead. We'll focus on those other options and skip the details of **IDynamicMetaObjectProvider**, which deserves an entire book.

Emulating Dynamic Objects with Dictionaries

Before we start building dynamic objects, let's talk about how you could use a dictionary to emulate a dynamic object with flexible members. The reason I mention this is two-fold. First, sometimes, a plain dictionary is more straightforward than a dynamic object. Second, most dynamic objects will use a dictionary or dictionary-like structure to represent themselves behind the scenes, so the pattern is helpful to understand.

The following creates a **Dictionary<string, object>** as an emulation of a dynamic object. The key acts as the name of the property, and the value is the contents of that property:

```
Dictionary<string, object> flexible = new Dictionary<string, object>();
flexible["Name"] = "George";
flexible["Age"] = 21;
```

You could imagine designing a class with a **Name** and **Age** property. That would be cleaner code, but in this case, we can add more things as we see fit as the program runs. You could not do that with a class.

Adding methods is more awkward, but a delegate (Level 36) can make this possible:

```
flexible["HaveABirthday"] = new Action(
      () => flexible["Age"] = (int)flexible["Age"] + 1);
```

Invoking this pseudo-method is also awkward, but it does work:

```
((Action)flexible["HaveABirthday"])();
```

This code retrieves the **Action** object from the dictionary, casts it to an **Action**, and then invokes it (the parentheses at the end).

We could even remove elements dynamically using the **Remove** method.

The syntax is inconvenient, especially for method calls, but it works. The other two approaches we will look at are more refined in this regard.

Using ExpandoObject

A second choice for a dynamic object is the **ExpandoObject** class in the **System.Dynamic** namespace. **ExpandoObject** is essentially just the dictionary approach we just saw, but with better syntax:

```
using System.Dynamic; // This namespace is not automatically included. Add it.

dynamic expando = new ExpandoObject();
expando.Name = "George";
expando.Age = 21;
expando.HaveABirthday = new Action(() => expando.Age++);

expando.HaveABirthday();
```

The syntax here is drastically improved. Adding properties is as simple as assigning a value to them. The syntax for adding and invoking a method got much better as well. For calling a method, it is identical to regular method calls! This cleaner syntax is because of that **dynamic** type and because **ExpandoObject** implements **IDynamicMetaObjectProvider** to define how its members should be found and used.

Interestingly, **ExpandoObject** implements **IDictionary<string, object>**, though it does so explicitly. If you cast an **ExpandoObject** to **IDictionary<string, object>**, you can use it as a dictionary to do things like enumerating all of its members or removing properties:

```
var expandoAsDictionary = (IDictionary<string, object>)expando;

foreach(string memberName in expandoAsDictionary.Keys)
    Console.WriteLine(memberName);

expandoAsDictionary.Remove("Age"); // Remove the Age property.
```

EXTENDING DYNAMICOBJECT

A second option for dynamic objects is to derive from the **DynamicObject** class. Deriving from **DynamicObject** is trickier than using **ExpandoObject**, but it also gives you much more control over the details. It is an abstract class with a pile of virtual methods that you can override. You use it by creating a derived class and overriding methods for any type of member you want to have dynamic access to. For example, you override **TryGetMember** if you want to dynamically get property values, **TrySetMember** if you want to set property values dynamically, and **TryInvokeMember** if you want to be able to invoke methods dynamically. Override each type of member that you want dynamic control over.

The example below is on the extreme simple end. It creates a dynamic object where properties and their **string**-typed values are supplied in the constructor and overrides **TryGetMember** and **TrySetMember** to allow users of the class to use those as properties:

```
public class CustomObject : DynamicObject
{
    private Dictionary<string, string> _data;

    public CustomObject(string[] names, string[] values)
    {
        _data = new Dictionary<string, string>();

        for (int index = 0; index < names.Length; index++)
            _data[names[index]] = values[index];
    }

    public override bool TryGetMember(GetMemberBinder binder, out object? result)
    {
        if (_data.ContainsKey(binder.Name))
        {
            result = _data[binder.Name];
            return true;
        }
        else
        {
            result = null;
            return false;
        }
    }

    public override bool TrySetMember(SetMemberBinder binder, object? value)
    {
        if (!_data.ContainsKey(binder.Name)) return false;

        // ToString(), in case it isn't already a string.
        _data[binder.Name] = value.ToString();
        return true;
    }
}
```

Using a dictionary, as shown here, is a common trend with dynamic objects.

The constructor is relatively straightforward, only taking the property names and their values and storing them in a dictionary.

The overrides for **TryGetMember** and **TrySetMember** are more interesting. Each has a **binder** parameter that supplies the information about what the calling code is trying to use. In particular, **binder.Name** is the specific name they are searching for. We use that in both **TryGetMember** and **TrySetMember** to look for a property with that name in the dictionary. If it exists, we return its associated value in **TryGetMember** and update it in **TrySetMember**. Both of these are expected to return whether the attempt to access the member was successful or not. In C#, a failure leads to a **RuntimeBinderException**.

With this object, you can dynamically use its properties:

```
dynamic item = new CustomObject(new string[] { "Name", "Age" },
                                new string[] { "HAL", "9001" });
Console.WriteLine($"{item.Name} is {item.Age} years old.");
```

TryGetMember and **TrySetMember** dynamically get and set properties. Override **DynamicObject**'s other members to get dynamic behavior for other member types, including methods (**TryInvokeMember**), operators (**TryUnaryOperation** and **TryBinaryOperation**), indexers (**TryGetIndex** and **TrySetIndex**), and more.

When to Use Dynamic Object Variations

C# is geared toward static typing. Choose static typing with regular classes and structs when you can. Don't forget that a dictionary is also a choice, and often a simpler one.

When using dynamic objects, use **ExpandoObject** when you can. It is the simplest to use. When that's not enough, derive a new class from **DynamicObject**.

Challenge Uniter of Adds 75 XP

"This city has used the Four Great Adds for a million clock cycles. But legend foretells a True Programmer who could unite them," the Regent of the City of Dynamak tells you. She shows you the four great adds:

```
public static class Adds
{
    public static int Add(int a, int b) => a + b;
    public static double Add(double a, double b) => a + b;
    public static string Add(string a, string b) => a + b;
    public static DateTime Add(DateTime a, TimeSpan b) => a + b;
}
```

"The code is identical, but the four types involved demand four different methods. So we have survived with the Four Great Adds. Uniting them would be a sign to us that you are a True Programmer." With dynamic typing, you know this is possible.

Objectives:

- Make a single **`Add`** method that can replace all four of the above methods using **`dynamic`**.
- Add code to your main method to call the new method with two **`int`**s, two **`double`**s, two **`string`**s, and a **`DateTime`** and **`TimeSpan`**, and display the results.
- **Answer this question:** What downside do you see with using **`dynamic`** here?

Challenge The Robot Factory 100 XP

The Regent of Dynamak is impressed with your dynamic skills and has asked for your help to bring their robot factory back online. It was damaged in the Uncoded One's arrival. Robots are manufactured after collecting their details, all of which are optional except for a numeric ID. After the information is collected, the robot is created by displaying the robot's details in the console. Here are two examples:

```
You are producing robot #1.
Do you want to name this robot? no
Does this robot have a specific size? no
Does this robot need to be a specific color? no
ID: 1
You are producing robot #2.
Do you want to name this robot? yes
What is its name? R2-D2
Does this robot have a specific size? yes
What is its height? 9
What is its width? 4
Does this robot need to be a specific color? yes
What color? azure
ID: 2
Name: R2-D2
Height: 9
Width: 4
Color: azure
```

In exchange, she offers the Dynamic Medallion and all robots the factory makes before you fight the Uncoded One.

Objectives:

- Create a new **`dynamic`** variable, holding a reference to an **`ExpandoObject`**.
- Give the dynamic object an **`ID`** property whose type is **`int`** and assign each robot a new number.
- Ask the user if they want to name the robot, and if they do, collect it and store it in a **`Name`** property.
- Ask if they want to provide a size for the robot. If so, collect a width and height from the user and store those in **`Width`** and **`Height`** properties.
- Ask if they want to choose a color for the robot. If so, store their choice in a **`Color`** property.
- Display all existing properties for the robot to the console window using the following code:

```
foreach (KeyValuePair<string, object> property in (IDictionary<string, object>)robot)
    Console.WriteLine($"{property.Key}: {property.Value}");
```

- Loop repeatedly to allow the user to design and build multiple robots.

LEVEL 46

UNSAFE CODE

Speedrun

- "Unsafe code" allows you to reference and manipulate memory locations directly. It is primarily used for interoperating with native code.
- You can only use unsafe code in unsafe contexts, determined by the **`unsafe`** keyword.
- Pointers allow you to reference a specific memory address. C# borrows the **`*`**, **`&`**, and **`->`** operators from C++.
- **`fixed`** can be used to pin managed references in place so a pointer can reference them.
- The **`stackalloc`** keyword allows you to define local variable arrays whose data is stored on the stack. A fixed-size array does a similar thing for struct fields.
- **`sizeof`** can tell you the size of an unmanaged type: **`sizeof(int)`**, **`sizeof(FancyStruct)`**.
- **`nint`** and **`nuint`** are native-sized integer types that compile differently depending on the architecture.
- You can invoke native/unmanaged code using Platform Invocation Services (P/Invoke).

A key feature of C# and .NET is that it manages memory for you (Level 14). But one of C#'s strengths is that it allows you to step out of this world and enter the unmanaged, "unsafe" world. Among other things, this makes it easy to work with code written in languages that do not use .NET, such as C or C++. Unmanaged code is sometimes called *native code*. Many C# developers will never touch unmanaged code, and even if you do, it likely won't be at the start of your C# journey. Yet C#'s ability to jump to the unmanaged world when conditions necessitate is a compelling reason to choose C# over other languages.

As a new C# programmer, the only lesson that matters now is knowing that this is possible. You do not need to come away with a deep mastery of unsafe or unmanaged code. If you are in a skimming or skipping mood, this level is a good one to do that on. The basics covered in this level will help shed light on how C# can interact with objects in memory that are not managed by the runtime and the garbage collector. This level is an overview of the basics. The ins and outs of unsafe code deserve an entire book.

Unsafe Contexts

Most C# code does not need to jump out of the realm of managed code and managed memory. However, C# does support certain "unsafe operations"—data types, operators, and other actions that allow you to reference, modify, and allocate memory directly. These operations are called *unsafe code*. Despite the name, it is not inherently dangerous. However, the compiler and the runtime cannot guarantee type and memory safety like they usually can. A less common but perhaps more precise name for it is *unverifiable code*.

Unsafe code can only be used in an *unsafe context*. You can make a type, method, or block of code an unsafe context using the **unsafe** keyword. This requirement ensures programmers use unsafe operations intentionally, not accidentally.

Making a block of code into an unsafe context is shown here:

```
public void DoSomethingUnsafe()
{
    unsafe
    {
        // You can now do unsafe stuff here.
    }
}
```

To make a whole method or every member of a type unsafe, apply the **unsafe** keyword to the method or type definition itself:

```
public unsafe void DoSomethingUnsafe()
{
}
```

And:

```
public unsafe class UnsafeClass
{
}
```

But even that is not enough. You must also tell the compiler to allow unsafe code into your program. This is typically done in the project's configuration file (the *.csproj* file). However, the easiest way to reconfigure a project like this is to just put an unsafe context in your code. When the compiler flags it, use the Quick Action to enable unsafe code in the project.

Pointer Types

In an unsafe context, you can create variables that are *pointer types*. A pointer contains a raw memory address where some data of interest presumably lives. The concept is nearly the same as a reference, though references are managed by the runtime, which sometimes moves the data around in memory to optimize memory usage. These are a different beast than both value types and reference types. The garbage collector manages references but not pointers.

You declare a pointer type with a ***** by the type:

```
int* p; // A pointer to an integer.
```

You can create a pointer to any *unmanaged type*. An unmanaged type is essentially any value type that does not contain references. That includes all of the numeric types, **char**, **bool**, enumerations, and any struct that does not have a reference-typed member, as well as pointers (pointers to pointers).

C# borrows three operators from C++ for working with pointer types: The address-of operator (**&**) for getting the address of a variable, the indirection operator (*****) for dereferencing a pointer to access the object it points to, and the pointer member access operator (**->**), for accessing members such as properties, methods, etc. on a pointer type object. These are shown below:

```
int x;
unsafe
{
   // Address-Of Operator. Gets the address of something and returns it.
   // This gets the address of 'x' and puts it in 'pointerToX'.
   int* pointerToX = &x;

   // Indirection Operator: Dereferences the pointer, giving you the object at
   // the location pointed to by a pointer. This puts a 3 in the memory location
   // pointerToX points at (the original 'x' variable).
   *pointerToX = 3;

   // Pointer Member Access Operator: allows access to members through a pointer.
   pointerToX->GetType();
}
```

This code illustrates how to use pointers, but if it weren't for a desire to show that, a simple **x = 3;** and **x.GetType();** would have been much cleaner.

Fixed Statements

You may need to get a pointer to some part of a managed object. This is possible but requires some work. Remember, the runtime and garbage collector manage reference types. They may move data around from one memory location to another as needed. Since a pointer is a raw memory address, we cannot allow a pointer's target to shift out from under us. A **fixed** statement tells the runtime to temporarily *pin* a managed object in place so that it doesn't move while we use it.

Suppose we have a **Point** class with public fields like this:

```
public class Point
{
    public double X;
    public double Y;
}
```

To get an instance's **X** or **Y** field requires pinning it to get a pointer to it:

```
Point p = new Point();

fixed (double* x = &p.X)
{
    (*x)++;
}
```

The garbage collector will not move **p** while that **fixed** block runs, ensuring our pointer will continue referring to its intended data.

A **fixed** statement demands declaring a new pointer variable; you cannot use one defined earlier in the method. A **fixed** statement can declare multiple new variables of the same type by separating them with commas: **fixed (double* x = &p.X, y = &p.Y) { ... }**

Stack Allocations

C# arrays are reference types. A variable of an array type holds only a reference, while the data lives on the heap somewhere. This behavior is not just tolerable but desirable in nearly all situations. But when needed, you can ask the program to allocate an array local variable on the stack instead of the heap with the **`stackalloc`** keyword:

```
public unsafe void DoSomething()
{
    int* numbers = stackalloc int[10];
}
```

You can only do this in an unsafe context, only for local variables, and only for unmanaged types. When the **`stackalloc`** line is reached, an additional 40 bytes (4 bytes per **`int`** for 10 **`int`**s) will be allocated on the stack for this method. When the code returns from **`DoSomething()`**, this memory is freed automatically when the method's frame on the stack is removed. It will not require the garbage collector to deal with it.

Fixed-Size Arrays

When working with code from unmanaged languages, such as C and C++, you sometimes want to share entire data structures. A complication arises when a struct holds an array with a reference to data that lives elsewhere. The struct's data is not contiguous, making it impossible to share with unmanaged code. Consider this struct with an array reference:

```
public struct S
{
    public int Value1;
    public int Value2;
    public int[] MoreValues;
}
```

The alternative is a *fixed-size array* or *fixed-size buffer*, which must always be the same size, but that stores its data within the struct instead of elsewhere on the heap:

```
public unsafe struct S
{
    public int Value1;
    public int Value2;
    public fixed int MoreValues[10];
}
```

This struct will hold all of its data together, with no references pointing elsewhere.

The runtime does not do index bounds checking on these arrays, as it does for regular arrays. You could access **`MoreValues[33]`** without throwing an exception, accessing random memory. Going past the end of an array can cause serious problems, hence the name "unsafe."

The sizeof Operator

The **`sizeof`** operator allows you to refer to the size in bytes of an unmanaged type without having to do the math yourself:

```
byte[] byteArray = new byte[sizeof(int) * 4];
```

This type is a constant value for the built-in types like **`int`**, **`double`**, and **`bool`**. The compiler will even replace **`sizeof(int)`** with a **`4`** when it compiles. For these situations, you can use **`sizeof`** anywhere in your code. This is a convenient tool if you forget how big something is:

```
Console.WriteLine(sizeof(double));
```

The main use of **`sizeof`** is in unsafe code to help you compute the size of more complex objects. For the non-built-in types, this can only be used in an unsafe context. **`sizeof`** is especially useful when dealing with complex structs because their sizes are not always obvious. For example, **`sizeof(long)`** is **`8`** and **`sizeof(bool)`** is **`1`**, but what is **`sizeof( LongAndBool)`**?

```
struct LongAndBool
{
    long a;
    bool b;
}
```

It is 16, not 9! The system may add padding bytes to the beginning, middle, and end of the struct. The CPU typically deals with blocks larger than a single byte (64 bits or 8 bytes on a 64-bit machine), and lining up data on those boundaries with padding makes it more efficient. The **`sizeof`** operator reveals the actual size of the struct.

C# does provide the tools to let you explicitly layout the members of a struct in memory for the rare cases when you need it.

THE `NINT` AND `NUINT` TYPES

In C#, the basic integer types are always the same size. But in native code, sometimes, the size of an **`int`** depends on the hardware being used. An **`int`** might be 32 bits on a 32-bit machine and 64 bits on a 64-bit machine. To address this, you can use the **`nint`** (native int) or **`nuint`** (native uint) types. These are essentially integers that compile to different types depending on the hardware, which helps your C# code better align with the native code it is trying to call.

CALLING NATIVE CODE WITH PLATFORM INVOCATION SERVICES

Platform Invocation Services, or *P/Invoke* for short, allows your managed C# code to directly call native (unmanaged, non-.NET) code. It lets you call native code, including C and C++ libraries, as well as operating system calls. The managed world of C# and the unmanaged world are quite different from each other, which means conversion between the two and marshaling data across this boundary with P/Invoke can get complicated.

Here is a simple example. Let's say we have some C code that defines an **`add`** function that adds two integers together. In your C code, you would ensure this method is exported from your DLL (a topic for a C book, not a C# book). In your C# code, you could produce a wrapper around this function with the **`extern`** keyword and the **`DllImport`** attribute (Level 47):

```
public static class DllWrapper
{
    [DllImport("MyDLL.dll", EntryPoint="add")]
    internal static extern int Add(int a, int b);
}
```

The **`extern`** keyword indicates the body of the method is defined outside of your C# code. You don't supply a body at all, but just end the line with a semicolon. The **`DllImport`** attribute (Level 47) indicates the native library and method to use when **`Add`** is called. (**`EntryPoint`** is not required if the method names are an exact match.)

All **`extern`** methods must be static, and you generally do not want to make them public for security reasons.

A call to **`DllWrapper.Add(3, 4)`** would send those two **`int`** parameters over to the unmanaged library, invoke the native **`add`** method, and return the result.

This example is a trivial one; real examples tend to be much more complicated. Even just getting the signatures and configuring the **`DllImport`** attribute can be a massive headache. The website **http://www.pinvoke.net** can help get this right.

Knowledge Check **Unsafe Code** **25 XP**

Check your knowledge with the following questions:

1. **True/False.** Unsafe code is inherently dangerous.
2. What keyword makes something an unsafe context?
3. What keyword pins a reference in place?
4. How do you denote a type that is a pointer to an **`int`**?

Answers: (1) False. **(2)** **`unsafe`**. **(3)** **`fixed`**. **(4)** **`int*`**.

LEVEL 47

OTHER LANGUAGE FEATURES

Speedrun

- **`yield return`** produces an enumerator without creating a container like a **`List`**.
- **`const`** defines compile-time constants that can't be changed.
- Attributes let you apply metadata to types and their members.
- Reflection lets you inspect code while your program is running.
- The **`nameof`** operator gets a string representation of a type or member.
- The **`protected internal`** and **`private protected`** accessibility modifiers are advanced accessibility modifiers that give you additional control over who can see a member of a type.
- The bit shift operators let you play around with individual bits in your data.
- **`IDisposable`** lets you run custom logic on an object before being garbage collected.
- C# defines a variety of preprocessor directives to give instructions to the compiler.
- You can access command-line arguments supplied to your program when it was launched.
- Classes can be made **`partial`**, allowing them to be defined across multiple files.
- The (dangerous) **`goto`** keyword allows you to jump to another location in a method instantly.
- Generic variance governs how a type parameter's inheritance affects the generic type itself.
- Checked contexts will throw exceptions when they overflow. Unchecked contexts do not.
- Volatile fields ensure that reads and writes happen in the expected order when accessed across multiple threads.

This is the final level that deals directly with C# language features. It covers various features that didn't have a home in other levels. Some are small and don't deserve a whole level to themselves. Others are relatively big but rarely used, so they weren't worth many pages in this book.

ITERATORS AND THE YIELD KEYWORD

The concept of **`IEnumerable<T>`** and **`IEnumerator<T>`** is straightforward: present items one at a time until you reach the end of the data. Most of the time, when **`IEnumerable<T>`** pops up, it is because you are using some existing collection type, such as arrays, **`List<T>`**, or **`Dictionary<T>`**. *Iterators* are another way to create an **`IEnumerable<T>`**. While you could implement the **`IEnumerable<T>`** interface (it is small), many C# programmers find iterators easier. An iterator is a special method that will produce or "yield" values one at a time. Here, the word "yield" means "to produce or provide," as you might see in an agricultural context. The following is an iterator method, which produces an **`IEnumerable<T>`** of the numbers 1 through 10:

```
IEnumerable<int> SingleDigits()
{
    for (int number = 1; number <= 10; number++)
        yield return number;
}
```

This returns an **`IEnumerable<int>`**, which you can use in a **`foreach`** loop:

```
foreach (int number in SingleDigits())
    Console.WriteLine(number);
```

Iterator methods are evaluated lazily. It does not produce the entire collection of items instantly. Only enough code in **`SingleDigits`** will run to encounter the next **`yield return`** statement as items are requested. When the **`foreach`** loop needs the first item, this method will run just enough code to reach the **`yield return`**. The next time a number is needed, execution within the method will resume where it left off and go until **`yield return`** is reached a second time. This repeats until all ten numbers have been produced and the **`SingleDigits`** method ends, or the **`foreach`** loop quits asking for more numbers.

You can have any logic in an iterator, including multiple **`yield return`** statements.

A **`yield break;`** statement will cause the sequence to end without getting to the method's closing curly brace.

Iterators do not have to ever complete. You can generate a sequence of items indefinitely. For example, this code will generate the sequence -1, +1, -1, +1, ... forever:

```
IEnumerable<int> AlternatingPattern()
{
    while (true)
    {
        yield return -1;
        yield return +1;
    }
}
```

There will always be more items to generate, no matter how far you go. And notably, this doesn't take up any memory because the items are produced on demand.

On the other hand, trouble is lurking if you attempt to materialize the entire **`IEnumerable<int>`** into a list or expect a **`foreach`** loop to find the end. This runs out of memory:

```
List<int> numbers = AlternatingPattern().ToList();
```

And this code will never terminate:

```
foreach (int number in AlternatingPattern())
    Console.WriteLine(number);
```

Async Enumerables

An async enumerable lets us combine iterator methods with the tasks we learned about in Level 44. Suppose you have a method that returns the contents of a website such as this:

```
public async Task<string> GetSiteContents(string url) { ... }
```

By making an iterator method with the return type **`IAsyncEnumerable<T>`** (in the **`System.Collections.Generic`** namespace), you can **`yield return`** an awaited task:

```
public async IAsyncEnumerable<string> GetManySiteContents(string[] manyUrls)
{
    foreach (string url in manyUrls)
        yield return await GetSiteContents(url);
}
```

The magic happens when you use an **`IAsyncEnumerable<T>`** with a **`foreach`** loop:

```
string[] urls = new string[] { "http://google.com", "http://amazon.com",
                               "http://microsoft.com" };

await foreach (string url in GetManySiteContents(urls))
    Console.WriteLine(url);
```

This code has the complexity of tasks, iterators, and **`foreach`** loops combined, but it allows you to process the results as they start coming back without waiting for the entire collection.

Constants

A *constant* (*const* for short) is a variable-like construct that allows you to give a name to a specific value. Perhaps the definitive example of this is the constant **`PI`** defined in the **`Math`** class, which has a definition that looks something like this:

```
public static class Math
{
    public const double PI = 3.1415926535897931;
}
```

A constant is defined with the **`const`** keyword, along with a type, a name, and a value. The value of a constant must be computable by the compiler, which means most constants will use a literal value, as **`PI`** does above. But you could also define a constant like this:

```
public const double TwoPi = Math.PI * 2;
```

Since **`Math.PI`** is a constant, the compiler can use it when computing **`TwoPi`** at compile time.

Constants are usually named with UpperCamelCase, but a few people like CAPS_WITH_UNDERSCORES. **`PI`** is an example of the second. Most of the Base Class Library uses the first convention, so **`PI`** is an inconsistency.

Despite their appearance, constants are not variables. You cannot assign values to them, aside from what the compiler gives it. They are static by nature, so you do not and cannot mark them **`static`** yourself.

How do constants compare with a **static readonly** variable?

The compiler replaces usages of a constant with the constant's value as it compiles the code. That means **double x = PI;** is turned into **double x = 3.1415926535897931;** by the compiler. The variable loses its association with the original constant. Often, this is not a big deal. There are, however, a few scenarios where this ends up causing problems. The general rule is to only use constants for things that will never change. **Math.PI** is a good example since the numeric value for π will never change. In other scenarios, a **static readonly** variable is usually preferable. Because a constant does not need to be looked up as the program is running, a constant can be slightly faster, which might also be a deciding factor in rare circumstances.

In general, the flexibility of **readonly** outweighs any other advantage of a constant. For example, you can define a **readonly** variable statically or one per instance. You can also assign values that must be computed at runtime, such as assigning a new instance, which cannot be done with a constant.

ATTRIBUTES

Compiled C# code retains a lot of rich information about the code itself. You can add to that richness by using *attributes*, which attach metadata to different parts of your code. Tools that inspect or analyze your code, including the compiler and the runtime, can access this metadata and adapt their behavior in response. For example:

- You mark a method as obsolete by applying an attribute. When you do this, the compiler will see it and emit warnings or errors when somebody uses the outdated code.
- You can mark methods as test methods, which a testing framework can run to ensure your code is still doing what you expected it to do.
- You can apply attributes to a class's properties that allow a file writing library to automatically dig through the object in memory and save it without writing custom file code for each object by hand.

Let's show how to apply attributes with that first example. The **Obsolete** attribute indicates that a method is outdated and should not be used anymore. In a small program, you would just delete the method and fix any code that uses it. It may take a while to clean everything up in a large program, so marking it obsolete can be a step in a long journey to eliminate it.

To apply the **Obsolete** attribute to an outdated method, place the attribute above the method inside of square brackets:

```
[Obsolete]
public void OldDeadMethod() { }
```

Attributes are like placing little notes on the different parts of the code. They survive the compilation process, so tools working with your code can see these attributes and use them. The compiler notices the **Obsolete** attribute and produces compiler warnings in any place where **OldDeadMethod** is called.

Many attributes have parameters that you can set. The **Obsolete** attribute has two: a **string** to display as an error message and a **bool** that indicates whether to treat this as an error (**true**) or a warning (**false**).

```
[Obsolete("Use NewDeadMethod instead.", true)]
public void OldDeadMethod() { }
```

Configured like this, you would see an error with the message "Use NewDeadMethod instead."

You can use multiple attributes in either of the following two ways:

```
[Attribute1]
[Attribute2]
public void OldDeadMethod() { }
```

Or:

```
[Attribute1, Attribute2]
public void OldDeadMethod() { }
```

Attributes on Everything

The attributes above are applied to a method, but you can use them on almost any code element. They are frequently applied to a class or other type definition.

```
[Obsolete]
public class SomeClass { }
```

In most cases, attributes are placed immediately before the code element they are for, but in some cases, there is no obvious "immediately before" spot, or that spot is ambiguous. There are special keywords that you can put before the attribute to make it clear in these cases. For example, this explicitly states that the **`Obsolete`** attribute is for the method (the default):

```
[method: Obsolete]
public void OldDeadMethod() { }
```

This one applies a (fictional) attribute to the return value of a method:

```
[return: Tasty]
private int MakeTastyNumbers() { /* What even is a tasty number? */ }
```

Each attribute decides which kind of code elements it can be attached to. Not every attribute can be attached to every type of code element. For example, the **`Obsolete`** attribute cannot be applied to parameters or return values, but it can be used on almost everything else.

Attributes are Classes

Attributes are just classes derived from the **`Attribute`** class. Their names typically also end with the world **`Attribute`**. For example, the **`Obsolete`** attribute we used above is officially called **`ObsoleteAttribute`**. You can use the attribute's full name (**`[ObsoleteAttribute]`**), or you can leave off the **`Attribute`** part (just **`[Obsolete]`**) if it ends with that.

The parameters you supply translate directly to a constructor defined in the attribute class. The following uses a two-parameter constructor:

```
[Obsolete("Use NewDeadMethod instead.", true)]
```

If an attribute has public properties, you can also set those by name:

```
[Sample(Number = 2)]
```

Attributes are usually created by people who want to work with your compiled code, including the people making the compiler, the .NET runtime, and other development tools. They decide what metadata they need, design the attribute classes that will give them that metadata, and then provide you with documentation that tells you how to use them.

But making attributes isn't hard either: make a new class based on **`System.Attribute`**:

```
[AttributeUsage(AttributeTargets.Constructor, AllowMultiple = true)]
public class SampleAttribute : Attribute
{
    public int Number { get; set; }
}
```

Notice how you use attributes when defining attributes! Now you can apply this **`Sample`** attribute to constructors:

```
[Sample(Number = 2)]
[Sample(Number = 3)]
public Point(double a, double b) { ... }
```

REFLECTION

Compiled C# code retains a lot of rich information about the code. This rich information allows a program to analyze its own structure as it runs. This capability is called *reflection*.

There are many uses for this. For example, you could use reflection to search a collection of DLLs to find things that implement an **`IPlugin`** interface, then create instances of each to add to your program. Or you could use reflection to find all the public properties of an object and display them all without knowing the object's type ahead of time.

Reflection is a broad topic that we can't cover in-depth here, but we can explore some practical examples that might pique your interest.

Most of the types involved in reflection live in the **`System.Reflection`** namespace. If you're doing much with reflection, you will probably want to add a **`using`** directive (Level 33) to make your life easier.

The **`Type`** class is the beating heart of reflection. It represents a compiled type in the system. An instance of the **`Type`** class represents the metadata of a specific type in your program. There are a few ways to get a **`Type`** instance. One is to use the **`typeof`** operator:

```
Type type = typeof(int);
Type typeOfClass = typeof(MyClass);
```

Or, if you have an object and want the **`Type`** instance that represents its type, you can use the **`GetType()`** method:

```
MyClass myObject = new MyClass();
Type type = myObject.GetType();
```

The **`Type`** class has methods for querying the type to see what members it has. For example:

```
ConstructorInfo[] contructors = type.GetConstructors();
MethodInfo[] methods = type.GetMethods();
```

Those return objects that represent each constructor or method of the type. If you want a specific constructor or method, you can use the **`GetConstructor`** and **`GetMethod`** methods, passing in the parameter types (and the method name for **`GetMethod`**):

```
ConstructorInfo? constructor = type.GetConstructor(new Type[] { typeof(int) });
```

```
MethodInfo? method = type.GetMethod("MethodName", new Type[] { typeof(int) });
```

The first line will find a constructor with a single **int** parameter. The second line will find a method in the type named **MethodName** with a single **int** parameter. If there isn't a match, the result will be null in both cases.

With a **ConstructorInfo** object, you can create new instances of the type:

```
object newObject = constructor.Invoke(new object[] { 17 });
```

With a **MethodInfo** object, you can invoke the method with a specific instance:

```
method.Invoke(newObject, new object[] { 4 });
```

The syntax is far worse than the natural equivalent:

```
MyClass newObject = new MyClass(17);
newObject.MethodName(4);
```

It is also not as efficient, nor can the compiler protect you from making mistakes. For those reasons, if you can do something without reflection, you should do so. But reflection is a valuable tool when the situation is right.

THE NAMEOF OPERATOR

The **nameof** operator lets you use the names of code elements from within your code. Consider this method:

```
void DisplayNumbers(int a, int b) =>
                       Console.WriteLine($"a={a} and b={b}");
```

Now let's say you rename the variables:

```
void DisplayNumbers(int first, int second) =>
                       Console.WriteLine($"a={first} and b={second}");
```

The output is now misleading. The names are no longer **a** and **b**, though the text still says they are. The **nameof** operator helps you get this right by producing a string based on the name of some code element:

```
void DisplayNumbers(int first, int second) =>
     Console.WriteLine($"{nameof(first)}={first} and {nameof(second)}={second}");
```

NESTED TYPES

You can define types within classes and structs. For example, a class could define two enumerations and another class within it. These are called *nested types*. These are mainly used for small types that support their container type. For example, a class might rely heavily on an enumeration, a record, or a small utility class. These could be defined inside the main class instead of as independent classes.

As a member of another type, these nested types can be (and often are) private. The containing class can still use a private nested class, but the rest of the program can't. An example is this **Door** class, which uses a private nested **DoorState** enumeration within the class:

```
public class Door
{
    private enum DoorState { Open, Closed, Locked }

    private DoorState _doorState = DoorState.Closed;

    public bool IsOpen => _doorState == DoorState.Open;
    public bool IsLocked => _doorState == DoorState.Locked;
}
```

Because **DoorState** is private, it can only be used within the class. A private field uses this type, but nothing public exposes it as a return type or parameter type.

Nested types do not need to be private. They are also sometimes protected, internal, or even public. If you need to use a nested class outside of the class they are defined in, you do so via the containing type name:

```
// Outside of 'Door', assuming 'DoorState' is made public.
if (door.State == Door.DoorState.Open) { ... }
```

You can nest types as deeply as necessary, but try to avoid more than one level of nesting.

Even More Accessibility Modifiers

We have seen four accessibility modifiers in this book:

- **public** is visible anywhere.
- **private** only visible in its containing class.
- **protected** is only visible in its containing class and derived classes.
- **internal** is visible anywhere in the containing project, but not other projects.

While **public** and **private** are two extremes, **protected** and **internal** grant visibility based on two very different aspects. The **protected** modifier works along inheritance lines, while the **internal** modifier works along project organization lines. In the few sporadic cases where you must concern yourself with both aspects, there are two other accessibility levels.

Making something **private protected** (both keywords) makes it visible only in derived types in the same project. It is more restrictive than **protected** and thus uses **private protected**.

Using **protected internal** (both keywords) makes it accessible in all derived classes and throughout the project—either is sufficient to grant access.

For both of these, the keywords can appear in either order.

Bit Manipulation

Most of the time, we work with data at a level higher than single bits. We use bundles of them for integers, Boolean values, strings, etc. But each is a container for a pile of bits.

C# provides tools for working with data as a raw pile of bits. To use them effectively, you must understand how types represent their data at the bit level. Getting into all of the details is too deep for this book. If you are new to programming, don't feel like you need to master this section before continuing. You will someday want to dig in and learn more about this, but you can treat this as a preview and a taste of what you can do.

Dealing with data at the bit level can open the door for more compact representations of our data. Few situations demand bit-level management, but it comes up when storage, memory, or network bandwidth is a scarcer resource than processing power. It is generally more work for the CPU to perform, and it always leads to source code that is harder to understand.

As an example of space savings, each **`bool`** uses up an entire byte when a single bit could be enough. If we have eight **`bool`** values, we could compact them into a single byte, reserving one bit for each true or false value (a 1 or a 0, respectively) and save 87.5% of the space used.

Consider the original Nintendo (NES) controller. This controller has eight buttons: A, B, Start, Select, Up, Down, Left, and Right. Each button can either be pressed or not. In a game that has to process input from this controller, we may decide we need a compact representation of the controller's state at any point in time. It is easy to imagine building a struct with eight **`bool`** fields, but if we want it to be compact, we can squeeze it into a single byte and assign one bit to each button's state:

Bit Number	Purpose	Sample
0	Up	1
1	Down	0
2	Left	0
3	Right	0
4	A	1
5	B	0
6	Start	0
7	Select	0

Because bits on the right end are usually the smaller valued bits, let's say we order these bits with bit #7 on the left end and bit #0 on the right end. This decision is arbitrary, but it is essential to be clear about the order. We don't accidentally flip the order somewhere. Writing the bit pattern for the sample would be **`00010001`**. Because bit #0 on the right end is a **`1`**, we interpret that as a pressed Up button. Bit #4 is also set (contains a **`1`**), representing a pressed A button. Thus, this bit pattern represents a controller with the player pressing both Up and A simultaneously and nothing else.

Bitshift Operators

We will need additional tools to write C# code that can work with bits.

The first pair of operators we will see is the bit shift operators. These take a bit pattern and move every bit to the left or right some number of spots. The left bit shift operator is **`<<`** and the right bit shift operator is **`>>`**. They are arrows that point in the direction the bits will move.

To illustrate, here is code that uses these two operators:

```
int controllerState = 0b00010001;
int shiftedLeft = controllerState << 2;
int shiftedRight = controllerState >> 3;
```

Remember, a literal value that starts with **`0b`** is a binary literal. This shows the exact bits that are in use. **`shiftedLeft`** will contain a new bit pattern with all of the bits moved two spots to the left. **`00010001`** becomes **`01000100`**. **`shiftedRight`** will contain the bit pattern with all bits moved three spots to the right. **`00010001`** becomes **`00000010`**. Note that bits can drop off the end if shifted far enough, and they are filled in with a **`0`** on the other end.

There are many ways we can use these bit shift operators, but here is a convenient trick:

```
int up    = 0b1 << 0; // bit #0 (00000001)
int down  = 0b1 << 1; // bit #1 (00000010)
int left  = 0b1 << 2; // bit #2 (00000100)
int right = 0b1 << 3; // bit #3 (00001000)
```

The code **`0b1 << 3`** (or **`1 << 3`**, since the bit pattern for **`1`** is also **`00000001`**) allows you to create a bit pattern that has a single bit changed to a **`1`**, and the number after the left bit shift operator indicates which bit number.

While we talked about using a single byte, I should point out that this code used **`int`**. We don't need 32 bits for that NES controller, just 8. However, all of these bit-level operators work on **`int`** and not **`byte`**. Fear not; if you want a single byte, you can cast the results to a **`byte`**.

Bitwise Logical Operators

The second group of operators for bit manipulation are the bitwise logical operators: **`&`** (bitwise-and), **`|`** (bitwise-or), **`~`** (bitwise-not or bitwise-complement), and **`^`** (bitwise-exclusive-or). These perform logical operations (the same category as **`&&`**, **`||`**, and **`!`**) but at the bit level, treating **`0`**'s as **`false`** and **`1`**'s as **`true`**. Each of these goes down the bits one by one, computing a result from the two inputs' corresponding spots. For example, to compute the correct result for bit #2, the operator uses bit #2 in both inputs to determine a result.

- The **`&`** (bitwise-and) operator produces a **`1`** (true) if both inputs are **`1`**. It produces **`0`** otherwise.
- The **`|`** (bitwise-or) operator produces a **`1`** if either input is a **`1`**.
- The **`^`** (exclusive-bitwise-or) operator produces a **`1`** if either input is a **`1`** but produces a **`0`** (false) if both are a **`1`**.
- The **`~`** (bitwise-not or bitwise-complement) operator is a unary operator and produces the opposite of whatever the bit was (a **`0`** becomes a **`1`**, a **`1`** becomes a **`0`**).

There are many potential uses for these operators, but consider the effect of the **`|`** operator in the following:

```
int aPressed =      0b1 << 4;                  // 00010000
int startPressed =  0b1 << 6;                  // 01000000
                                               // --------
int combined = aPressed | startPressed;        // 01010000
```

The **`|`** operator has the effect of producing a new value equivalent to both A and Start buttons being pressed. We can use **`|`** to turn any individual bit to a **`1`**. That's a technique we can use whenever a button is pressed.

The **`^`** operator would have a similar effect, except that it would toggle the bit instead of turning it on.

The **`&`** operator also has an interesting effect. Consider the following two uses:

```
int downButton      = 0b00000010;
int controllerState = 0b01000010; // Down and Start
int isDownPressed   = controllerState & downButton; // 0b00000010
```

And:

```
int downButton      = 0b00000010;
int controllerState = 0b01000001; // Up and Start
int isDownPressed   = controllerState & downButton; // 0b00000000
```

Using **`&`** like this results in one of two possibilities: all **`0`**'s or the down button pattern. That means we could use the following to see if some controller state has a specific button pressed:

```
int downButton      = 0b00000010;
int controllerState = 0b01000010; // Down and Start
bool isDownPressed  = (controllerState & downButton) == downButton;
```

This result is a **`bool`**, not a bit pattern.

Finally, we can turn off a specific bit by using the **`&`** and **`~`** operators:

```
controllerState = controllerState & ~downButton;
```

The **`~`** operator will produce the inverse bit pattern, so **`~downButton`** becomes **`11111101`**. When combined with any other value, it will produce a result where most bits remain what they were but force the single bit to **`0`**, turning it off.

All six bit-based operators have compound assignment operators: **`<<=`**, **`>>=`**, **`&=`**, **`|=`**, **`~=`**, and **`^=`**. These work just like **`+=`** and the other compound assignment operators.

Flags Enumerations

Remember, enumerations are integers behind the scenes but with better names for each number. Our controller problem could be well suited to an enumeration:

```
[Flags]
public enum Buttons : byte
{
    Up =     1 << 0, // 00000001
    Down =   1 << 1, // 00000010
    Left =   1 << 2, // 00000100
    Right =  1 << 3, // 00001000
    A =      1 << 4, // 00010000
    B =      1 << 5, // 00100000
    Start =  1 << 6, // 01000000
    Select = 1 << 7  // 10000000
}
```

This code uses two advanced features of enumerations: assigning numbers to each member and choosing **`byte`** as the underlying type. The first ensures that each button is correctly assigned to its specific bit. The second keeps the size to a single byte.

This code also added the **`[Flags]`** attribute, which gives enumeration values a slick **`ToString()`** representation, while also stating that these values are specifically meant to be one item per bit. When single bits are used to represent bool values, they are frequently referred to as *flags*, and programmers will talk about bits being *set* or *raised* if it is a **`1`** or *unset* or *lowered* if it is a **`0`**. With this enumeration defined, we can do stuff like this:

```
Buttons state = Buttons.Up | Buttons.A; // Indicates that Up and A are pressed.
```

With the **`[Flags]`** attribute, you can call **`state.ToString()`** and get the following output:

```
Up, A
```

Without **[Flags]**, the integer value (17) that corresponds to 00010001 is displayed instead.

You can use all of the other bit-based operators with this **Buttons** enumeration, but you may also find the **HasFlag** method useful:

```
bool aButtonIsPressed = state.HasFlag(Buttons.A);
```

USING STATEMENTS AND THE IDISPOSABLE INTERFACE

The garbage collector can clean up any heap memory that it manages. Things get sticky when some of those objects have their fingers on some sort of unmanaged object or memory. This is a surprisingly common occurrence. For example, **FileStream** and **StreamWriter** (Level 39) use native operating system objects to function.

When something accesses unmanaged resources, it must provide a way for the outside world to ask it to release those resources. Implementing the **IDisposable** interface is the way to do this. **IDisposable** has a single **void Dispose()** method. When you detect that it is time for such an object to clean itself up, you call this method. (If you forget, the garbage collector can recognize something is **IDisposable** and call it for you, but it is better if you do it yourself.)

You will want to watch for objects that implement **IDisposable** and dispose of them correctly. These often appear when you need something outside your program to help you, such as file access and operating system and network requests.

Let's look at how you might call **Dispose**. An imperfect but simple version may look like this:

```
FileStream stream = File.Open("Settings.txt", FileMode.Open);
while (stream.ReadByte() > 0)
    Console.WriteLine("Read in a byte.");
stream.Close();
stream.Dispose();
```

What this solution is missing is that **Dispose** will not be called on **stream** if an exception is thrown (Level 35). A better way would be to use a **finally** block:

```
FileStream stream = null;
try
{
    stream = File.Open("Settings.txt", FileMode.Open);
    while (stream.ReadByte() > 0)
        Console.WriteLine("Read in a byte.");
    stream.Close();
}
finally
{
    stream?.Dispose();
}
```

Whether we leave the **try** block by reaching its end naturally, returning early, or throwing an exception, the **finally** block will always run. By calling **Dispose** inside the **finally**, we can be confident it will get called. But we can do even better. The following shows a **using** statement (different from **using** directives at the top of your files), which is a shorter way to do what we just did:

```
using (FileStream stream = File.Open("Settings.txt", FileMode.Open))
{
    while (stream.ReadByte() > 0)
        Console.WriteLine("Read in a byte.");
    stream.Close();
}
```

This approach is shorter and less error-prone than writing it out yourself, so you should use it when you can.

We often don't care strongly about when a **using** block ends, and if that is the case, we can make it even shorter:

```
using FileStream stream = File.Open("Settings.txt", FileMode.Open);

while (stream.ReadByte() > 0)
    Console.WriteLine("Read in a byte.");

stream.Close();
```

Here, the **using** block ends at the end of the method. But this spares us the extra curly braces and indentation, making for much simpler code.

using statements are convenient but only work for an **IDisposable** local variable. If we have an **IDisposable** field, we should make our class implement **IDisposable**, and call our field's **Dispose** method from there.

Many C# programmers will also use **IDisposable**'s **Dispose** method to unsubscribe from events. Merely implementing **IDisposable** does not eliminate event leaks, but it does make a convenient way to detach an object from any events it has tied itself to.

Keep an eye out for types that implement **IDisposable** and dispose of them when done.

Preprocessor Directives

SIDE QUEST

You can embed specific instructions to the compiler directly in your C# code using *preprocessor directives.* These all start with the **#** symbol. Some of the more interesting ones are described below.

#warning and #error

The two simplest preprocessor directives are the **#warning** and **#error** directives. These tell the compiler to emit a compiler warning or error on that line, showing the associated message:

```
#warning Enter whatever message you want after.
#error This text will show up in the Errors list if you try to compile.
```

While simple, these have limited uses. For example, forcing an error is not very valuable because it ensures you will never have working code. These are sometimes combined with other compiler directives for more practical results.

#region and #endregion

The second pair of preprocessor directives is **#region** and **#endregion**. These mark the beginning and end of a block of code and allow you to give the section a label:

```
#region The region where AwesomeClass is defined.
public class AwesomeClass
```

```
{
    // ...
}
#endregion
```

The compiler ignores regions, but Visual Studio and other IDEs include regions in the editor feature called *code folding*. You can see small boxes with + and - symbols to the left of the code. Clicking on these will hide that section of code. Most IDEs will give you foldable sections for methods, types, and even things like **`try`**/**`catch`** blocks and loops. Regions marked with **`#region`** and **`#endregion`** will also be foldable.

Some programmers use regions to organize parts of their code. They might have a **`#region Fields`** and a **`#region Constructors`**. Regions can be handy if done right, though my personal experience shows that this is hard to do. One of three things usually happens. In some instances, the type is small enough that **`#region Everything`** adds more clutter than value. Other times, people accidentally (or lazily) put things in the wrong region, making them misleading. In other cases, regions mark parts of a class that you really ought to extract into its own class and object. There are plenty of C# programmers who dislike (even hate) regions. Use them as you see fit, but make sure they serve their purpose well.

You can nest regions, but every **`#region`** must have a matching **`#endregion`**.

Working with Conditional Compilation Symbols

The compiler can use *conditional compilation symbols* (or *symbols*) to decide what parts of a code file to include. These symbols are either present or not, though a C# programmer would say that the symbol is *defined* or not. The Debug configuration defines the **`DEBUG`** symbol, while the Release configuration does not. (These symbols are usually written in ALL_CAPS.) Using these symbols, we can tell the compiler to include a section of code or not based on whether a symbol is defined by using **`#if [SYMBOL]`** and **`#endif`**. For example:

```
Console.WriteLine("Hello ");
#if DEBUG
Console.WriteLine("World!");
#endif
```

If you compile this program in the Debug configuration, which defines the **`DEBUG`** symbol, it will be as though your file looked like this:

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

But if you compile this program in the Release configuration, which does not define the **`DEBUG`** symbol, it will be as though your file looked like this:

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

This allows your code to compile in different ways for different situations. Using symbols for different configurations (like **`DEBUG`**) is common, as are symbols for different operating systems (**`#if WINDOWS`** or **`#if LINUX`**).

You should keep conditional compilation like this to a minimum. The more you do, the more configurations you must test before a release.

There is also **`#else`** and **`#elif`**, which are like **`else`** and **`else if`** in C# code:

```
#if WINDOWS
Console.WriteLine("Hello Windows!");
```

```
#elif LINUX
Console.WriteLine("Hello Linux!");
#else
Console.WriteLine("Hello mystery operating system!");
#endif
```

The **`#if`** and **`#elif`** directives can also check multiple symbols with **`&&`** and **`||`**.

Symbols are typically defined in the project's configuration file (*.csproj*), but **`#define`** can allow you to define a specific symbol for a single file:

```
#define WINDOWS
```

Similarly, **`#undef`** undefines a symbol for a single file:

```
#undef DEBUG
```

The **`#define`** and **`#undef`** directives must come at the start of a file.

Command-Line Arguments

SIDE QUEST

Main methods have a **`string[] args`** parameter. Using top-level statements, you can access this variable in your main method even though you can't see it anywhere. If you explicitly write out a **`Main`** method (the traditional model described in Level 33), this parameter is explicitly written out. **`args`** contain command-line arguments supplied by somebody launching your program from the command line. Command-line arguments allow somebody to dictate what your program should do without needing an interactive back-and-forth via **`Console.WriteLine`** and **`Console.ReadLine`**. It makes it easy for people to run your program from a script or without constant interaction from a human.

You can access these command-line arguments through the **`args`** parameter, which are all strings and may need parsing:

```
int a = Convert.ToInt32(args[0]);
int b = Convert.ToInt32(args[1]);

Console.WriteLine(a + b);
```

We could run this program from the command line like this:

```
C:\Users\RB\Documents>Add.exe 3 5
```

Many programs throughout history have produced an **`int`** value to indicate success or failure. You can also change **`Main`** to return an **`int`** if you want to emulate this. If you do, the age-old tradition is that returning **`0`** indicates success, and anything else indicates failure, with specific numbers representing specific types of errors.

Partial Classes

SIDE QUEST

You can split a definition of a class, struct, or interface across multiple files or sections with the **`partial`** keyword:

```
public partial class SomeClass
{
    public void DoSomething() { }
}

public partial class SomeClass
{
    public void DoSomethingElse() { }
}
```

This separation allows two things (like the programmer and an automatic code generator) to work in their parts of the class without fear of breaking what the other is doing. For example, with GUI applications, you often use a designer tool to drag and drop controls on the screen. The designer can edit one part of the class in response, while the programmer can manually edit a second part without interfering with each other. The designer will never break what you wrote, and you won't hurt what the designer generated.

Partial Methods

With partial classes, one part sometimes needs to rely on a method it expects to be defined in another part. Partial methods let one part define a method signature without a body while hoping another part will fill in the rest:

```
public partial class SomeClass
{
    public partial void Log(string message);

    public void DoStuff() => Log("I did stuff.");
}

public partial class SomeClass
{
    public partial void Log(string message) => Console.WriteLine(message);
}
```

The top portion knows the **`Log`** method will exist and puts it to use, but the definition is provided in another part. If the method's body is never defined, the compiler will catch it and produce a compiler error.

If the partial method is **`void`** and implicitly private (no stated accessibility level), the compiler will let you skip the definition if you want. Instead, it strips out calls to the undefined method. Because the method is both private and **`void`**, this removal has no consequences.

The Notorious goto Keyword

SIDE QUEST

C# has a **`goto`** statement that lets you jump to arbitrary spots within a method. Teaching the correct usage of a **`goto`** statement is easy: don't! That is a small amount of hyperbole, but it is the best advice for a new C# programmer. (Feel free to skip ahead now.)

Many constructs in C# will cause the flow of execution to jump around from place to place. An **`if`** statement, loops, **`continue`**, **`break`**, and **`return`** all do so. But these all have a particular structure that makes it easy for programmers to analyze and understand. The **`goto`** statement does not. In truth, **`goto`** usually leads to code so convoluted that it is almost impossible to remove the **`goto`** while being confident the logic remains the same.

Yet, the mechanics are simple enough. Within a method, you can place a *label* or a *labeled statement*. Elsewhere in the method, you can place a **`goto`** statement, which calls out the label to jump to when reached. For example:

```
    int number = 0;

Top:
    Console.WriteLine(number);
    number++;
    if (number < 10)
        goto Top;
```

Top is a label, and **goto Top;** is a **goto** statement that will cause the flow of execution to jump back to it. This code, however, is mechanically the same as this code:

```
for (int number = 0; number < 10; number++)
    Console.WriteLine(number);
```

Which is easier for you to understand?

This code is as simple **goto** gets, and it can be far worse. Even straightforward logic tends to become a quick mess with a **goto**. If you're considering a **goto**, exhaust all other possible scenarios first. Usually, some other arrangement of the code cleanly solves the problem better.

Generic Covariance and Contravariance

SIDE QUEST

With inheritance, you can substitute the derived class any time the base class is expected. Suppose we have the following two classes in a small inheritance hierarchy:

```
public class GameObject // Any object in the game world.
{
    public float X { get; set; }
    public float Y { get; set; }
}

public class Ship : GameObject
{
    public string? Name { get; set; }
}
```

Because of inheritance, you can use a **Ship** object anywhere a **GameObject** is expected:

```
GameObject newObject = new Ship();
```

Or, with a method whose signature is **void Add(GameObject toAdd)**, you can call it with **Add(new Ship())**.

A **Ship** is substitutable for a **GameObject** because it is just a special kind of **GameObject**.

This substitutability does not automatically transfer to generic types that use them. **Ship** is derived from **GameObject**, but **List<Ship>** is not derived from **List<GameObject>**. Here is why this would be problematic:

```
List<GameObject> objects = new List<Ship> { new Ship(), new Ship(), new Ship() };
objects.Add(new Asteroid());
```

The type for **objects** is a **List<GameObject>**, but it currently contains a **List<Ship>**, only able to hold **Ship** instances. On the second line, we try to add a new **Asteroid** object. This seems reasonable because the variable's type is **List<GameObject>**. But **objects** currently references the more specific **List<Ship>**, which cannot handle **Asteroid**s. That is why generics do not automatically support an inheritance-like relationship.

While the general case prohibits a universally applied inheritance-like relationship, certain specific situations give us a glimmer of hope. Imagine if **List<T>** only used **T** objects as outputs and never as inputs—that is, we never used it as a parameter type or in a property setter, etc., and only used it for return types and property getters. The problematic situation we described cannot exist.

If a type parameter is only used for outputs (return types and property getters), you can mark the type parameter as such and support this inheritance-like structure. **IEnumerable<T>** meets these conditions with its T parameter. It has marked **T** with the **out** keyword:

```
public interface IEnumerable<out T> { /* ... */ }
```

Since **IEnumerable**'s **T** parameter is marked with **out**, the following is allowed:

```
IEnumerable<Ship> ships = new List<Ship>();
IEnumerable<GameObject> objects = ships;
```

objects will hold a reference to a **List<Ship>**. This is close to the dangerous scenario we looked at earlier, but through the **IEnumerable<T>** interface, we cannot attempt to add other subtypes to it. We're safe.

You can only apply **out** to generic type parameters on interfaces and delegates, not on a class or struct. When a type parameter has **out** on it, the compiler ensures it is never used as an input (parameter or property setter).

Rules that dictate how generic types work concerning their type parameters are called *variance* rules. As we saw with **List<T>** initially, the default is *invariance* (or you could say that it is *invariant*), meaning that there is no relationship at all. When used only as an output, as we saw with **IEnumerable<out T>**, it is called *covariance* (or you could say that **IEnumerable<T>** is *covariant* with respect to **T**).

The opposite also exists and is called *contravariance,* or being *contravariant*. Suppose an interface uses a generic type parameter only for inputs. In that case, you can place the **in** keyword on the generic type parameter:

```
public interface IStringMaker<in T>
{
    string MakeString(T value);
}
```

You could make the following two implementations of this:

```
public class ShipStringMaker : IStringMaker<Ship>
{
    public string MakeString(Ship ship) => $"Ship named {ship.Name}.";
}

public class GameObjectStringMaker : IStringMaker<GameObject>
{
    public string MakeString(GameObject g) => $"Object at ({g.X}, {g.Y}).";
}
```

Because **IStringMaker<T>** is contravariant, you can do this:

```
IStringMaker<Ship> stringMaker1 = new ShipStringMaker();
IStringMaker<Ship> stringMaker2 = new GameObjectStringMaker();

Console.WriteLine(stringMaker1.MakeString(new Ship { Name = "USS Enterprise" }));
Console.WriteLine(stringMaker2.MakeString(new Ship { Name = "USS Enterprise" }));
```

This compiles and displays:

```
Ship named USS Enterprise.
Object at (0, 0).
```

Contravariance may feel a little backward; it takes some getting used to. But keep in mind that what we see above for the **`GameObjectStringMaker`** is the equivalent of calling the following method with a **`Ship`** instance, which works fine:

```
public string MakeString(GameObject g)
```

Checked and Unchecked Contexts

SIDE QUEST

Unlike math, integer types on a computer eventually run out of bits to represent large numbers. As we saw in Level 7, we overflow the type when we push beyond a type's limits. The typical result is wrapping around. For example:

```
int x = int.MaxValue;
Console.WriteLine(x + 1);
```

`int.MaxValue` is 2147483647, so mathematically, this should display 2147483648. Instead, it displays -2147483648. Most of the time, careful selection of your integer types addresses this issue. If you are pushing the limits of a type, upgrade to the next biggest type.

If overflow is unacceptable in a situation, the alternative is to throw an exception when overflow occurs (Level 35). You can get the system to do this instead by performing math operations in a *checked context*. The simplest way to define a checked context is like this:

```
int x = int.MaxValue;
Console.WriteLine(checked(x + 1));
```

Placing an expression in parentheses after the **`checked`** keyword will ensure everything in the expression will occur in a checked context. Alternatively, if you want multiple statements done in a checked context, you can use curly braces instead:

```
checked
{
    Console.WriteLine(x + 1);
}
```

Checking for overflow like this slows things down. You want to do it only in places where there is a legitimate concern, usually only a few lines here or there. If you need a much broader checked context, you can turn it on for an entire project. Open the project's properties by right-clicking on it in the Solution Explorer, going to the **Build** tab, clicking **Advanced**, and checking the box labeled **Check for arithmetic overflow/underflow**.

If you are in a checked context and want to escape it temporarily, you can use the **`unchecked`** keyword to do the opposite:

```
int x = int.MaxValue;
Console.WriteLine(unchecked(x + 1));
```

Checking for overflow applies only to the integer types; floating-point numbers just become infinity instead.

Volatile Fields

SIDE QUEST

C# generally guarantees that instructions happen in the order they are written. This order is called *program order*. However, sometimes, the compiler or even hardware itself may adjust the timing of writing to memory locations for performance reasons. This is called *out-of-order execution*. For a single thread, these optimizations still ensure that program order is respected for practical purposes. But with multiple threads, these optimizations may cause surprising behavior. Imagine that two threads share access to a **`private int _value`** field and a **`private bool _complete`** field and run the following code. Thread 1 is running this:

```
_value = 42;
_complete = true;
```

Thread 2 is running this:

```
while (!_complete) { }
Console.WriteLine(_value);
```

You would expect Thread 2 to display 42 in all scenarios. Still, with these optimizations, Thread 2 may see the change to **`_complete`** before seeing **`_value`**'s change, and it could move past the **`while`** loop to the output of **`_value`** before it becomes **`42`**, displaying its previous value.

If you declare a **`volatile`** field, the compiler and hardware will not make out-of-order optimizations, and everything works as you would expect. Memory writes before **`_complete`** will be visible to all threads before the write to **`_complete`** happens, so Thread 2 won't see them updated out of order. This is done by adding the **`volatile`** keyword to the field:

```
private volatile bool _complete;
```

Making fields **`volatile`** should only be done when there is a clear need. Otherwise, it is just slowing your code down. And remember that there is a bigger arsenal of synchronization tools at your disposal, including the **`lock`** statement.

Knowledge Check — Other Features — 25 XP

Check your knowledge with the following questions:

1. **True/False.** The **`const`** keyword is equivalent to the **`readonly`** keyword.
2. What is the name of the class that lets you inspect (reflection) a type's definition at runtime?
3. What keyword allows you to jump to a named location elsewhere in the method?
4. What keyword will enable you to split a class's definition into multiple parts?
5. Name two bit manipulation operators.
6. **True/False.** An enumeration definition can contain a class definition.
7. **True/False.** A class definition can contain an enumeration definition.
8. What preprocessor directives begin and end a section of **`DEBUG`**-only code?
9. What keyword is involved when automatically cleaning up objects that implement **`IDisposable`**?
10. **True/False.** You can create never-ending sequences with the **`yield`** keyword.

Answers: (1) False. **(2) `System.Type`**. **(3) `goto`**. **(4) `partial`**. (5) **`<<`**, **`>>`**, **`&`**, **`|`**, **`^`**, **`~`**. **(6)** False. **(7)** True. **(8) `#if DEBUG`** and **`#endif`**. **(9) `using`**. **(10)** True.

LEVEL 48

BEYOND A SINGLE PROJECT

Speedrun

- A solution can have more than one project in it.
- Code in a project only has access to things contained in itself or things it references.
- To build a multi-project solution, add references between the projects.
- Projects can also add dependencies on compiled *.dll* files or NuGet packages.

OUTGROWING A SINGLE PROJECT

There will come a time as you are building a C# program where you will suddenly realize, "I shouldn't have to write this code again. It already exists! I want to reuse it!" It may already exist because you wrote it last week. Or maybe it already exists because six programmers on the other side of the planet decided it was a good idea and built something for the world to reuse.

By default, your programs have access to everything in your project's code and the Base Class Library that comes with C#. When you need something more, you will need to configure your projects to know about the additional code.

This need usually comes in one of three flavors: (1) a suite of closely related programs with overlapping functionality, (2) a massive program where you want reusable components as byproducts, and (3) you find code that does some specialized task that you want to reuse instead of making from scratch. We'll focus on the first and third scenarios here, though the techniques you use for the second scenario are essentially the same as for the first.

Let's imagine you have been building a space-based role-playing game (RPG) game with your own board-short-wearing astronaut character named SpaceDude, who ends every sentence with "dude." You have slaved away for months and have a **`DudeConsole`** class that automatically appends "dude" onto any string:

```
public class DudeConsole
{
    public static void WriteLine(string message) =>
            Console.WriteLine($"{message}, dudes!");
    public static void Write(string message) =>
            Console.Write($"{message}, dude");
}
```

And of course, your main method for SpaceDudeRPG looks like this:

```
DudeConsole.WriteLine("It's time to level up");
```

You're ready to branch out and build your second game in the Space Dude universe, but this time as a real-time strategy (RTS) game instead of an RPG. For this second game, there is overlap with the first. This game doesn't need to know anything about leveling up, but Space Dude still ends every sentence with "dudes," and it would be nice to reuse `DudeConsole` without copying and pasting the class into the new project.

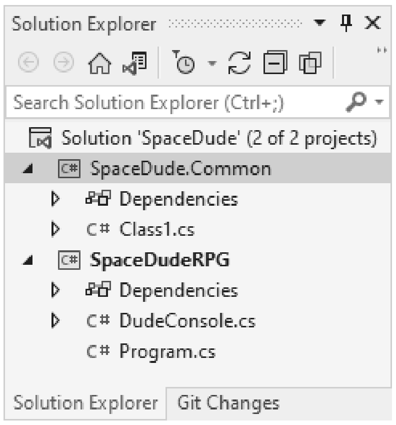
This situation is where multiple projects can come in handy. Everything in a project is compiled into a single assembly—either a single *.dll* file or *.exe* file. If we place all of the reusable code into one project and then have separate projects for the things unique to SpaceDudeRPG and SpaceDudeRTS (a total of three projects), both games can reuse the shared project.

In this scenario, we have not begun working on SpaceDudeRTS, only SpaceDudeRPG. SpaceDudeRPG likely contains just a single project with both unique and reusable things.

Step 1: Creating the shared project. We create a second project in our solution to contain this reusable code. You can do this through Visual Studio's menu under **File** > **New** > **Project**. This opens the new project dialog, which we have seen when creating a brand new program from scratch.

A project meant for reuse rather than a complete application is a *code library*, *class library*, or simply a *library*. Libraries do not have an entry point. They just contain type definitions.

In the new project dialog, you will pick an appropriate project type for the class library. The right choice may depend on what you are doing, but the template called **Class Library** is a good default choice.

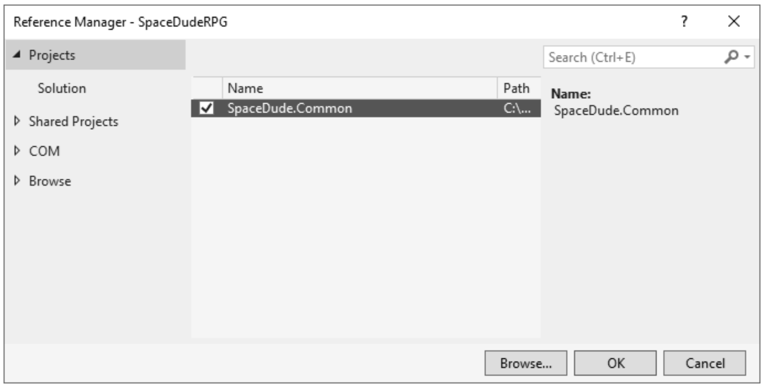


After selecting the project type, you will fill in its details. Give it a name (maybe `SpaceDude.Common`?) and in the Solution dropdown, choose **Add to solution**. After completing this step, your solution will contain two projects instead of one, as shown on the right.

Step 2: Move the shared code to the new project. We can drag and drop *DudeConsole.cs* to the new project, but the default behavior is copying files, not moving them. Hold Shift while dragging, and *DudeConsole.cs* will be moved instead. The template also placed a *Class1.cs* file in there. We can delete that.

Step 3: Cleanup namespaces. At this point, you may want to clean up namespaces if you're using them (Level 33). Code meant for reuse should usually be in a namespace. Namespaces usually mirror project and folder structures, so `DudeConsole` may be moving from the `SpaceDudeRPG` namespace to the `SpaceDude.Common` namespace.

Step 4: Link the projects. By default, a project can access everything in the Base Class Library and anything in the project itself. We just moved a file between projects, so it is no longer accessible in the `SpaceDudeRPG` project. If we want one project to know about another project's contents, we have to configure it that way. This is called adding a *dependency*, or you might say that we want to add `SpaceDude.Common` as a dependency of `SpaceDudeRPG`. To add `SpaceDude.Common` as a dependency, right-click on `SpaceDudeRPG` in the Solution Explorer and choose **Add > Project Reference**. This opens the Reference Manager, allowing you to add all kinds of dependencies, including the kind we want. If you select the **Projects** tab on the left side, you will see a list of other projects in the solution, including `SpaceDude.Common`. By each project is a checkbox, which you can check to add the dependency. Once you're done, you can press the **OK** button and close the dialog.



You can confirm that the project was added by expanding `SpaceDudeRPG`'s Dependencies node in the Solution Explorer. The new dependency should show up under the Projects node. After doing this, the project will gain access to the code in its new dependency. You can repeat this for any other dependency that you want to add.

Step 5: Use the shared project code. At this point, you can start using the code from the shared project. If you moved a class that you were using and changed namespaces, you may need to update (or add) `using` directives.

Dependencies are one-way. `SpaceDudeRPG` knows about `SpaceDude.Common`, but `SpaceDude.Common` does not know about `SpaceDudeRPG`. The system enforces that. You cannot create two projects that reference each other (or three that form a loop). That is necessary because Visual Studio must figure out what to build first. There must be a definite ordering for which projects to compile first, and circular dependencies would prevent that.

Dependency structure can be as complex as necessary. A project can have a thousand dependencies or a chain of dependencies a thousand links long.

So how do we go about letting the new `SpaceDudeRTS` game use `SpaceDude.Common`? We could add `SpaceDudeRTS` as a third project in the same solution. Building the solution would then compile both games. This configuration is more common for situations like building the iOS and Android versions of the same game, but it could work here as well.

A second approach is to make a new solution for `SpaceDudeRTS` and then add `SpaceDude.Common` as an existing project to the solution. Do this by choosing **File > Add > Existing Project** from the menu, then browsing to find the *.csproj* file for the shared project. `SpaceDude.Common` would show up in each solution, and changing the code in one would affect the other. That can be problematic because you may not realize you just broke the other game with a change in the common code. The other game's unique aspects are out of sight in another solution. A second limitation is that everybody on your team will need matching folder structures.

A third approach is to treat the shared code as an independently deployable package (see the NuGet section later in this level) and have each of the other two games reuse the compiled *.dll* instead of sharing its source code. This is the gold standard if you can manage building, testing, and deploying shared libraries.

NuGet Packages

Having a project depend on compiled code in the form of a *.dll* is often cleaner than gathering and recompiling all source code from scratch every time. C# projects can reference a compiled code library in the form of a plain *.dll* (open the Reference Manager dialog, hit the Browse button, and find the *.dll* file). However, a more popular approach is to use NuGet.

NuGet is a *package manager*. Package managers are a type of program that knows about reusable components called *packages*. You can think of a package as a combination of a *.dll* file and metadata about the *.dll*, most notably its version number. Each NuGet package is in a *.nupkg* file (essentially a *.zip* file containing the *.dll* and the metadata). Using NuGet, you can configure a project to use some specific version of a package.

We could take our `SpaceDude.Common` project and turn it into a NuGet package. We would work on it until we have a version worth releasing, then put it in the NuGet package. We put this package on a NuGet server or host, allowing other projects to reference it when they want to reuse it.

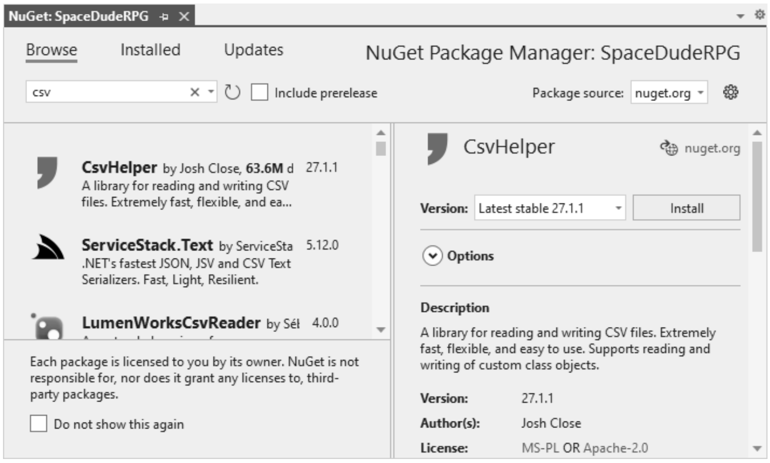
The most popular NuGet host is the website **nuget.org**. This is a public server full of hundreds of thousands of unique packages. If you intend to release something publicly and freely, this is the place to publish it. `SpaceDude.Common` is probably not something we want to share publicly, so we would avoid putting it there and find an alternative solution. There are many other NuGet hosts out there, and NuGet also supports deploying them to the file system on a computer your whole team has access to. The details of NuGet hosting are beyond this book, but NuGet is a pretty straightforward system.

You will eventually find value in making and publishing NuGet packages as you build larger programs. For now, it will be far more common to use existing NuGet packages than to post your own.

The website **nuget.org** has a package for almost everything. If you think someone else might have solved it before, you're usually right, and nuget.org contains their solution. For instance, in Level 39, we learned about the CSV file format. A search for "CSV" finds 1033 packages related to CSV. One of these is bound to help. For example, the `CsvHelper` package looks far superior to our hand-made version in virtually every way.

Using a NuGet package often gives us something that instantly works better and in less time than doing it ourselves. The drawback is that we don't control the code and can't easily change the parts we don't like. Plus, we need to learn how to use the classes and other types contained in the library. Most widely used NuGet packages have plentiful documentation and sample code, so this usually isn't painful.

Adding a NuGet package to a project is easy. Right-click on the project in the Solution Explorer and choose **Manage NuGet packages**. This opens a tabbed window that lets you see what NuGet packages are currently added to a project, search for new packages to add to the project, and point out which NuGet packages have an update you could upgrade to:


NuGet: SpaceDudeRPG
Browse
Installed
Updates
NuGet Package Manager: SpaceDudeRPG
csv
Include prerelease
Package source: nuget.org
CsvHelper by Josh Close, 63.6M d 27.1.1
A library for reading and writing CSV files. Extremely fast, flexible, and ea...
ServiceStack.Text by ServiceSta 5.12.0
.NET's fastest JSON, JSV and CSV Text Serializers. Fast, Light, Resilient.
LumenWorksCsvReader by Sé 4.0.0
Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.
Do not show this again
CsvHelper
nuget.org
Version: Latest stable 27.1.1
Install
Options
Description
A library for reading and writing CSV files. Extremely fast, flexible, and easy to use. Supports reading and writing of custom class objects.
Version: 27.1.1
Author(s): Josh Close
License: MS-PL OR Apache-2.0

The **Browse** tab lets you add NuGet packages. You can search to find the package you want, review its license, dependencies, and other metadata, and install it on that same page. The **Installed** tab lets you check what packages the project currently depends on. You can use either tab to uninstall or update a NuGet package.

A project with a dependency on a NuGet package can use the code contained in it. You will likely need to add **`using`** directives or fully qualified names to take advantage of it.

The **`internal`** accessibility modifier deserves to be revisited in light of this level. If something is **`public`**, other projects will be able to use it. When you make code libraries like this, that is often precisely your intent. But if there are things in a project that are merely details of how the project gets its job done and not meant for reuse, those things should be **`internal`**. If something is **`internal`**, you know you can change it without breaking somebody else's code.

Challenge Colored Console 100 XP

The Medallion of Large Solutions lies behind a sealed stone door and can only be unlocked by building a solution with two correctly linked projects. This multi-project solution is a key that unseals the door.

Objectives:

- Create a new console project from scratch.
- Add a second Class Library project to the solution.
- Add a static class, **`ColoredConsole`**, to the library project.
- Add the method **`public static string Prompt(string question)`** that writes **`question`** to the console window, then switches to cyan to get the user's response all on the same line.
- Add the method **`public static void WriteLine(string text, ConsoleColor color)`** that writes the given text on its own line in the given color.
- Add the method **`public static void Write(string text, ConsoleColor color)`** that writes the given text without a new line in the given color.
- Add the right references between projects so that the main program can use the following code:

```
string name = ColoredConsole.Prompt("What is your name?");
ColoredConsole.WriteLine("Hello " + name, ConsoleColor.Green);
```

Challenge The Great Humanizer 100 XP

The people in the village of New Ghett come to you with a complaint. "Our leaders keep giving us **`DateTime`**s that a hard to understand." They show you an example:

```
Console.WriteLine($"When is the feast? {DateTime.UtcNow.AddHours(30)}");
```

This code displays things like the following:

```
When is the feast? 12/21/2021 9:56:34 AM
```

"We keep showing up too early or too late for the feasts! We have to pull out our clocks and calendars! Isn't there a better way?" When pressed, they describe what they'd prefer. "What if it said **`When is the feast? 11 hours from now`** or **`When is the feast? 6 days from now`**? This is easier to understand, and we wouldn't show up too early or too late. If you can do this for us, we can retrieve the NuGet Medallion for you."

You know that writing code to convert times and dates to human-friendly strings would be a lot of work, but you suspect other programmers have already solved this problem and made a NuGet package for it.

Objectives:

- Start by making a new program that does what was shown above: displaying the raw **`DateTime`**.
- Add the NuGet package **Humanizer.Core** to your project using the instructions in the level. This NuGet package provides many extension methods (Level 34) that make it easy to display things in human-readable formats.
- Call the new **`DateTime`** extension method **`Humanize()`** provided by this library to get a better format. You will also need to add a **`using Humanizer;`** directive to call this.
- Run the program with a few different hour offsets (for example, **`DateTime.UtcNow.AddHours( 2.5)`** and **`DateTime.UtcNow.AddHours(50)`**) to see that it correctly displays a human-readable message.

Level 49

Compiling in Depth

Speedrun

- Computers only understand binary—1's and 0's. Data and instructions are both represented in binary.
- Binary instructions for a computer are unique to the circuitry of that specific computer.
- Binary is difficult for a human to work with. Assembly puts binary instructions into a human-readable form, which are turned into binary with an assembler.
- Programming languages move away from a one-to-one correspondence of machine instructions. Statements can be turned into many machine instructions by the language's compiler.
- Instruction set architectures (such as x86/x64 and ARM) standardize on specific machine instructions, making it easier to target multiple machines simultaneously.
- Virtual machines provide their own instruction set architecture and compile it down to the actual machine instructions as the program runs. They decouple knowledge about specific operating systems and instruction sets from the language.
- The virtual machine in .NET is called the Common Language Runtime (CLR).

Level 3 covered the compiler basics: transforming human-readable source code into machine-understandable binary instructions. That's a good start, but there is more that is worth knowing. That is the focus of this level.

Hardware

Early computing devices were mechanical or clockwork devices that could reliably compute answers to a specific question. For example, the 2000-year old Antikythera Mechanism calculated the sun's and moon's positions, allowing it to predict the strength of tides—valuable information for sailors.

More recent computers have programmable hardware. Rather than solving a single problem, a set of instructions (the program) is fed to the machine to activate different parts of the machine. This allows the computer to solve a wide variety of problems.

On modern computers, both the set of instructions and data itself are represented using physical (usually electrical) signals that are either present (indicated with a 1) or absent (indicated with a 0). Each signal can be in one of two states, making it a *binary* signal. A single signal, a 1 or a 0, is called a *bit*, short for "binary digit." Bits are put into groups of eight, called a *byte*.

The specifics of *how* a computer represents its data and how a computer interprets its instructions depend on the computer's hardware design. Taking binary instructions designed for one computer and feeding it to another does not (necessarily) work. This is a problem that we will revisit throughout this level.

On top of that, most modern computers don't run a program on the "bare metal." An operating system such as Windows or Linux runs on the computer, acting as a middleman between the hardware and the individual programs that need to run, guarding access to the hardware and providing services to the running programs. These operating systems play a role in determining how instructions are unpacked from storage and placed into the computer's central processing unit (CPU) to run. Even with identical hardware, two computers running different operating systems are not guaranteed to run the same.

Both hardware and operating system affect what can be done, and a compiler must account for these differences as it compiles.

Let's pick apart some sample binary. What follows is based on an actual binary representation, but there are many out there. Don't worry about the specifics; consider this an illustrative possibility. How binary instructions *could* work, rather than how it *must* work. The following is intended to perform 2+2:

```
00100100000010000000000000000010
00100100000010010000000000000010
00000001000010010101000000100000
```

At first glance, this probably makes little sense. Adding whitespace to separate logical groups helps a bit:

```
001001 00000 01000 0000000000000010
001001 00000 01001 0000000000000010
000000 01000 01001 01010 00000 100000
```

The first two lines are similar in structure. These each place a value of 2 into a memory location. The third line is structurally different. It does the actual addition.

The first group on the first two lines, **`001001`**, indicates "load immediate" instructions. This takes a specific value and places it into a memory location. The second group, **`00000`**, is ignored by load immediate instructions. The third group says where to store the value. **`01000`** on the first line refers to a spot in circuitry that the hardware guys call **`$t0`**. The **`01001`** on the second line refers to a location called **`$t1`**. The last block of bits, **`0000000000000010`**, is a representation of the number 2. So the first line says, "Put the value of 2 into **`$t0`**." The second line says, "Put the value of 2 into **`$t1`**."

The third line has a different structure. The first block, **`000000`**, indicates that this is one of the arithmetic operations. The sixth and final block, **`100000`**, indicates addition. The second and third blocks are the inputs to the addition. We saw these bit patterns before. Those indicate **`$t0`** and **`$t1`** again. The fourth block, **`01010`**, is where the result is placed: **`$t2`**. So this line says, "Take what is in **`$t0`** and **`$t1`**, add them, and save the results to **`$t2`**."

These 0's and 1's are tied to a specific computer. A different computer would not necessarily associate these bit patterns with 2+2, and a very different thing might happen if it isn't just gibberish to the hardware.

Assembly

You could program in binary, but not easily. However, even adding spaces made a difference. Let's take that idea further. We can use a text-based scheme instead of a binary scheme for representing each of these elements. (Once again, this is based on a real scheme, but it is meant to be illustrative and nothing more.) Instead of **`01000`**, we write **`$t0`**. Instead of **`0000000000000010`**, we write **`2`**:

```
li $t0, 2
li $t1, 2
add $t2, $t0, $t1
```

The text above is called an *assembly programming language*, or *assembly* for short. This assembly code is much easier to understand than its binary equivalent. The **`2`**'s and **`add`** stand out! Other parts are less obvious, like **`li`**, which is short for "load immediate." But if you have documentation, you could learn this language and be vastly more efficient than writing binary.

In assembly, each line is an exact match for a binary instruction, just written in a way that a human can read. It is still tied to the hardware itself, just like its binary equivalent.

While using assembly has made a programmer's job easier, it is meaningless to the computer. Since assembly is so closely tied to binary, it is not hard to build a special program—an *assembler*—that can take this text and translate it into binary.

Programming Languages

Once we begin separating what the human writes from the instructions the computer runs, the floodgates open. Why stop with simple transformations? Why not something more aggressive? For example, instead of **`add $t2, $t0, $t1`**, why not the following?

```
$t2 <- $t0 + $t1
```

Using **`<-`** and **`+`** make this operation much more intuitive.

Or how about this?

```
$t3 <- $t0 + $t1 + $t2
```

There is no three-way **`add`** instruction, but it is not hard to imagine how we could turn that into two separate additions. We are no longer tied to only the instructions the hardware provides. Some lines can be turned into many instructions. As long as we can build a program to translate for us, we can do whatever we want. We have just invented the programming language! As the translation from human-readable text to working binary instructions gets more sophisticated, we ditch the "assembler" name and use the term *compiler*.

While assembly code is an exact match for the machine's hardware, programming languages can transform a single statement into two, ten, or two hundred instructions. Many instructions can be written with little effort, and programmers can be more productive.

Not everybody agrees on the best human-readable starting point, and some representations are better than others for specific types of problems. These two points mean there is room for more than one programming language. Indeed, there are hundreds or thousands of widely used programming languages globally, each with its own syntax and compiler. C# is but one of many, though its power and ease of use make it one of the few well-used and well-loved languages. Over a long, productive programming career, you can and should learn others.

Instruction Set Architectures

Now that we can create code efficiently, let's look at the problem of every computer having potentially unique hardware and instruction sets. It would be a shame to need a different programming language for every computer.

Two techniques help alleviate this problem. They are the topics of this section and the next.

The first is *instruction set architectures*, often abbreviated to *ISA*, or just an *architecture*. ("Architecture" has many definitions in the computing world.) These architectures define a standard set of instructions that many computers use—even computers made by competitors. With a standardized set of instructions, a compiler can build binary code for all of them at one time. (Well, aside from the operating system differences.) And if you were designing a new computer, picking one of the standard architectures means you will be able to more readily use all of the code compiled for that standard architecture. It is a win-win.

There are many architectures, but two are by far the most common. Most cell phones use ARM, and most desktops and laptops use x86. The original x86 architecture was for 32-bit computers, so they extended x86 to handle 64-bit computers. This extension is called x86-64, or simply x64. If a compiler can target both ARM and x86/x64, it can run on almost every hardware platform in the world.

Of course, there is still the operating system to consider. Thus, when choosing a target to run on, it is typically a combination of architecture and operating system, for example, Windows 64-bit (x64) or Android ARM.

Virtual Machines and Runtimes

The other technique used to make it easier to build software that works everywhere is *virtual machines*. A virtual machine defines a "virtual" instruction set. No hardware has the circuitry to run this virtual instruction set, but instead, a software program—the virtual machine software—processes these instructions. C# uses this model. C#'s virtual machine is called the *Common Language Runtime*, or *CLR*. The virtual instruction set is called *Common Intermediate Language*, *CIL,* or *IL*. (It was once called Microsoft Intermediate Language, and the abbreviation MSIL still pops up in a few places.)

Because virtual instruction sets do not need to work in hardware, they tend to be far more capable than physical instruction sets. For example, IL understands the concepts of classes (Level 18), inheritance (Level 25), and exceptions (Level 35).

When the instruction set is this advanced, making language compilers is easy, and change can happen faster. Adding new languages to the ecosystem is also easy. Visual Basic and F#, among other languages, share the same IL instruction set. This also allows code written in these languages to be effortlessly reused in other languages. The entire standard library—the Base Class Library—is the same for all of these languages.

The CLR's job is to translate the virtual instructions into hardware instructions. This virtual machine compiles IL code into machine-executable instructions on a method-by-method basis, as they are first needed. Because the final compilation step happens when first needed, the special compiler inside the CLR is called the *Just-in-Time compiler*, the *JIT compiler*, or the *jitter*.

Thus, compilation happens in two steps. There are advantages to this separation, especially in an ecosystem with many different programming languages, as the world surrounding C# is. Adding new languages is easy because you don't have to consider all possible architectures and operating systems, just the virtual machine. Plus, the virtual instruction set is much more capable, so the effort of building a compiler is small. Existing languages trying to add new features have the same benefits. Most optimization happens in the JIT compiler, allowing a single optimization to make every involved language better at the same time.

Some people complain that JIT compilation makes C# programs slow. In truth, it is a second step to run, and that inevitably takes time. Most situations are hardly affected by this cost. Still, for those that are, C# provides an option to perform the JIT compilation when the main C# compiler runs (or immediately after). This is called *ahead-of-time compilation*, or *AOT compilation*. We'll see more about that in Level 51.

Knowledge Check **Compiling** **25 XP**

Check your knowledge with the following questions:

1. What are the two most popular instruction set architectures?
2. **True/False.** Binary is universal; nearly all computers use the same single set of instructions.
3. What is the name of C#'s virtual machine?
4. Name two programming languages besides C#.
5. What is the name of the virtual instruction set that C# compiles to?

Answers: (1) x86/x64 and ARM. **(2)** False. **(3)** Common Language Runtime (CLR). **(4)** C++, Java, Visual Basic .NET, F#, among many others. **(5)** Common Intermediate Language (CIL or IL for short).

Level 50

.NET

Speedrun

- .NET is the framework that your C# code builds on and runs within.
- There have been many .NET implementations (.NET Framework, .NET Core, Mono, etc.), but the latest is .NET 6.
- Common Intermediate Language is the virtual instruction set that your C# code compiles to. It serves as input to the Common Language Runtime, .NET's runtime.
- .NET provides a Base Class Library that gives you useful types and tools to help build your program.
- .NET also includes many app models, which are frameworks for making specific application types, such as desktop development (WinForms, WPF, UWP), web development (ASP.NET), smartphone app development (Xamarin Forms, MAUI), and game development (Unity, MonoGame, etc.)

.NET is a software framework built to make application development easy. It is the framework that your C# programs leverage. It is the runtime surrounding your running programs, providing the things the C# language promised to do for you. It is the virtual machine that transforms your code into something that the target machine can run directly. In short, it is the magic that envelops your program—and to a degree, even the programming experience—that makes your life easier. Here, we will get into more details about .NET. It is big enough that you will never stop learning more about it, but at this point, we can get into more depth than the early levels could.

The History of .NET

Most frameworks have only a single implementation. .NET has had many incarnations, and a basic understanding of its history will help you understand .NET and its future better.

In the beginning, Microsoft made the *.NET Framework*. The original. Some people felt it had two deal-breakers: it was closed source (only Microsoft could make it better), and it only worked on Windows. They made an alternative called *Mono* (Spanish for "monkey"), which was open source and ran on many operating systems. Over the years, there were several other flavors of .NET, but these two led the pack. The .NET Framework was the most popular and set the pace and direction, with Mono following behind, attempting to keep up while adding a few unique capabilities.

With many flavors of .NET floating around, the .NET world began to feel fragmented. To bring everything back into alignment, *.NET Core* was born. .NET Core is a Microsoft-supported, open-source flavor of .NET that runs on all major operating systems. It has quickly become the preferred flavor of .NET. Aside from some older programs that are hard to port, almost all new development happens in this world.

In November of 2020, an update of .NET Core was released, simply called *.NET*. The "Core" moniker is gone, and the flavor of .NET formerly known as .NET Core is now simply .NET to make it clear that this is the future of the .NET world.

The older .NET Framework and Mono versions still exist and can be used when needed, but this book assumes you're using .NET 6 or newer.

The Components of .NET

.NET is a vast software framework. It is made of three layers, each built on the one below it, with your program atop it all:

1. **Common Infrastructure.** The foundation, which includes the Common Language Runtime (the runtime your program executes within), CIL (.NET's intermediate language), and the SDK (the development tools used to build .NET programs).
2. **Base Class Library.** A library of reusable code that solves everyday problems.
3. **App Models.** These are libraries, frameworks, and deployment models for building specific application types, like websites, desktop apps, mobile apps, and games.

We'll discuss each in turn.

Common Infrastructure

.NET's common infrastructure is what everything else builds on. It is a collection of software programs and tools. We will discuss the most important ones below.

Common Intermediate Language

Perhaps the most foundational element of .NET is the Common Intermediate Language (CIL) that we saw in Level 49. It is a definition of a virtual instruction set. It unites each of the .NET languages and plays a vital role in ensuring they evolve fast enough to stay productive and relevant. You can think of CIL as an advanced, object-oriented assembly language shared by all .NET languages, including C#, Visual Basic, and F#.

The Common Language Runtime

The beating heart of .NET is its runtime: the *Common Language Runtime*, or the *CLR*. The CLR is the virtual machine and runtime of .NET. A runtime is additional code that runs with or around your program's code and fulfills the promises made by the language itself.

A runtime is compiled code that allows the language to do what the language designers promised it would do. One example is that you expect your program will begin running in its main method. The runtime makes sure that this is where things start.

Nearly every programming language has a runtime in one form or another. Some bake the runtime into the program, while others keep it separate. C# can support either approach.

The runtime does many important jobs for you, but these are perhaps the most important:

- **Just-in-time compilation.** This is the part of the runtime that makes it a virtual machine. It translates the CIL code produced by the C# compiler and turns it into instructions that the underlying physical hardware can run.
- **Memory management and garbage collection.** This is the single most significant task of the runtime. It maintains the managed heap (Level 14) and cleans up the memory that your program no longer needs.
- **Memory safety and type safety.** As a side effect of the runtime's job of managing memory and collecting garbage, the runtime knows a great deal about how your program uses memory. The runtime uses this information to ensure that your code does not inadvertently access memory that isn't officially used (*memory safety*) or attempt to treat a segment of memory as something it is not (*type safety*). These eliminate large categories of problems that can be hard to catch in other languages.

The .NET Software Development Kit

The .NET Software Development Kit (SDK) is a collection of tools available to you as you build .NET programs. This SDK is an amalgamation of many small, focused tools. Visual Studio and other IDEs build on top of the SDK to make programming life easier, but you could use the SDK directly if you wanted.

While you can install the .NET SDK separately, installing the correct components in Visual Studio will also automatically install the .NET SDK.

The heart of the .NET SDK is the command-line tool **`dotnet`**. With the SDK installed, you can use this tool to create, compile, test, and publish projects and more. With Visual Studio or another full-fledged IDE, you won't usually need to use the **`dotnet`** tool directly. If you're using Visual Studio Code or another lightweight text editor for programming, you will frequently use the **`dotnet`** tool.

Base Class Library

The second layer of .NET is the *Base Class Library* (*BCL*). The word *library* denotes a reusable code collection, and a *class library* is simply a library containing object-oriented classes. Some libraries are more broadly applicable than others. For any given language, the designers provide a library that will always be available to any program written in that language. This library is called a *standard library*. The Base Class Library is C#'s standard library. It contains things useful in nearly every program you could write.

The BCL is vast and grows with every version of .NET. We cannot cover all of it in this book (it would take volumes). Still, this book covers the most versatile types in the BCL, including the built-in types, collection types, types for timekeeping, exception types, delegate and event types, and types for asynchronous programming. As you continue programming in C#, you will encounter additional types in the BCL that you will find useful. That is part of the journey.

App Models

An *app model* is a framework (and usually a deployment model) for building a specific application type. While the Base Class Library contains things that are useful in virtually any application type, app models give you the tools to make specific application types. Each app model is vast and deserving of entire books. Between that and the fact that books about each app model assume you already know the basics of C#, I will avoid diving deep into these app models in this book. However, I have an article on this book's website, which identifies helpful resources for starting in each app model (**https://csharpplayersguide.com/articles/app-model-recommendations**).

Web App Models

ASP.NET (pronounced "A-S-P dot net") is a set of technologies designed to work together, aimed at web development in C#. Web development with ASP.NET is one of the most popular uses of C#.

There are three different tools for building the web pages themselves—the "front end." MVC is the oldest of the bunch but is widely used. Razor Pages is a streamlined version of MVC that is also popular. Blazor is the new kid on the block, letting you run C# directly in your clients' browsers using WebAssembly. All of these play nice with JavaScript, which allows you to tap into the entire web ecosystem.

On the server-side (the "back end"), there are several tools as well. Web API helps you build web services (REST services), SignalR does real-time communication with clients, etc.

Mobile App Models

Xamarin Forms lets you build mobile apps that work on iOS and Android with the same codebase. (Nearly all of it is shared. Each has unique elements on top.) The cross-platform nature of Xamarin Forms is its key selling point.

But Xamarin Forms is evolving into the *.NET Multi-platform App User Interface* (*.NET MAUI*). MAUI is not officially out as this book goes to print but should be released in 2022. MAUI supports not only iOS and Android but also macOS and Windows, and potentially Linux as well. It is hard to say what the future holds here, but MAUI may quickly become a powerful tool for native (non-web) application development.

Desktop App Models

Desktop app models make it easy to build traditional desktop applications. These applications tend to be workhorse-type applications that need a lot of power to do elaborate, focused work.

There are three different desktop app models in .NET, though all three are Windows-only. (But see the section on Xamarin Forms in the Mobile App Models section.)

Windows Forms (*WinForms*) has existed since the start of .NET. It is mostly in maintenance mode now, with no recent massive feature updates, but it has been a tried and true model for a long time. It is also perhaps the simplest desktop app model, so it can still be a good starting point for new UI programmers.

Windows Presentation Foundation (WPF) is newer than WinForms. I have often said (with a small dose of hyperbole) that WPF is twice as difficult and ten times more powerful than WinForms. It has a robust binding system that makes it easy to hook up the user interface to your C# objects and a powerful styling system that lets you make user interfaces that look how you want them. However, WPF is also mainly in maintenance mode, with primarily only performance, security, and bug fix updates.

The *Universal Windows Platform* (*UWP*) is the most modern desktop app model and frequently gets feature updates. Its approach is similar to WPF. Its initial launch was a bit bumpy and did not have the same level of adoption as other desktop app models, but it has been evolving quickly. If you want cutting-edge desktop development, this is a good choice.

As described in the *Mobile App Models* section above, after MAUI is released, it may also be a compelling choice for desktop applications.

Game Development

Because C# is both powerful and easy to use, it is a popular game development choice. There has been an explosion in C# game engines, frameworks, and libraries. There is something out there for anybody, regardless of your specific needs.

None of these are supported by Microsoft or the .NET Foundation, and in some cases, they target slightly older versions of .NET. Some of the features covered in this book may not be available quite yet.

The undisputed king of C# game development is the Unity game engine (**unity.com**). Unity is feature-rich, has a large community, and works essentially everywhere.

But Unity is not the only option. MonoGame (**monogame.net**) is another popular choice. It is less “opinionated” than Unity about how you should make your game, giving you more flexibility in how you build it, at the expense of needing to do more legwork. It can run in a wide variety of places and has an active community.

I could go on for pages listing game development tools—there are that many—but here are a few other notable choices: CryEngine (**cryengine.com**), Godot (**godotengine.org**), Stride3D (**stride3d.net**), raylib (**raylib.com**), and Ultraviolet Framework (**ultraviolet.tl**).

Knowledge Check .NET 25 XP

Check your knowledge with the following questions:

1. Which app model excites you the most and why?
2. Name two desktop UI frameworks in the .NET world.
3. What is the name of the extensive library that is available in all C# programs?
4. **True/False.** .NET is limited to only Windows.

Answers: (1) (depends). **(2)** WPF, Windows Forms, UWP. **(3)** Base Class Library. **(4)** False.

Level 51

Publishing

Speedrun

- The compiler uses configuration and source code to produce the final software.
- *.csproj* and *.sln* files contain project and solution configuration.
- Each build configuration defines its own settings and configuration. Debug and Release are the default configurations.
- Publishing your program compiles and assembles your whole program so that it can be deployed. There are many settings used to make your published artifacts precisely what you want.

Publishing your program is the process of taking your compiled program, putting it into the form you want to deliver it in, and making it available to users. The specifics depend heavily on what type of project you have made. A website is published to a web server, and users access it through their browser. A mobile app typically goes through a store like Google Play or Apple's App Store. A game might go through something like Steam. Each of these will have its own way (and instructions) to package your program for publishing.

However, the starting point is usually the same: compile your code and then package the compiled code and other files together. That is what we will focus on here.

Build Configurations

The first step is to build our code with the correct configuration. This isn't entirely new to us; we have been compiling code all along. When we compile, our source code is only part of the input to the compiler. The other part is settings and configuration. We could run the compiler ourselves (*csc.exe*) and supply all of the settings needed for each file and project. However, the usual approach (which Visual Studio does for us) is to ask a tool called *MSBuild* to compile our code. MSBuild uses *.sln* and *.csproj* files as configuration for each file, project, and the whole solution. MSBuild extracts the proper settings from these files and feeds them to the C# compiler.

These *.sln* and *.csproj* files can be edited in a text editor, though it is usually done through Visual Studio, which gives you a fancy graphical interface to avoid typos and errors.

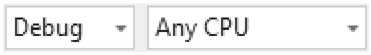
MSBuild allows you to define build configurations. Each build configuration has its own unique settings. When you compile your project or solution, you tell it which build configuration to use.

By default, most solutions and projects will have two build configurations: Debug and Release. Unless you change things, there are only two differences between these configurations. The Debug configuration does not optimize code, while Release does. Optimizing the code can make debugging (Bonus Level C) a bit harder, but it also makes it run faster. The Debug configuration also defines the DEBUG symbol (Level 47) while Release does not.

Modifying these two configurations is common. You can also create additional build configurations, though this is less common. Debug and Release are usually enough. You can also delete Debug and Release and replace them with something new, but this is rare. Other than convention, there is nothing special about these configurations.

When building, you also pick a specific architecture (Level 49) to target. The default is "Any CPU," which is agnostic to any particular architecture and should run anywhere. Most of the time, this is what you should use. After all, that is the purpose of the CLR in the first place. If you reference a native library that targets a specific architecture, you also need to restrict your own code to just that architecture.

Every time you compile, you choose which configuration and architecture to use. The current one is shown (and changeable) in the toolbar at the top of Visual Studio by the Run button:



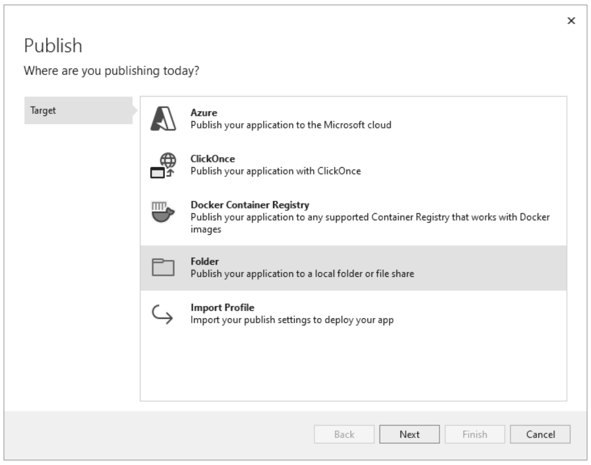
Each project can define settings for every build configuration and architecture (platform) pairing. That gives you control over how it should compile under any scenario.

You can also override which configuration and architecture each project uses for any build at the solution level. If you want to tweak these settings, you can edit the *.sln* file by hand (not recommended) or edit it through Visual Studio by right-clicking on the top-level solution node in the Solution Explorer and choosing Configuration Manager.

While it makes sense to use the Debug configuration while creating your program, you will usually want the Release configuration when you publish your program.

Publish Profiles

Once you have defined the needed build configurations (the defaults are a good starting point), you are ready to deploy. Rather than re-choosing the details every time you publish, you define a publish profile, which you can subsequently reuse. The easiest way to define these profiles is to right-click on the project to publish in the Solution Explorer and choose **Publish**. This opens the following dialog:


Publish
Where are you publishing today?

Target

Azure
Publish your application to the Microsoft cloud

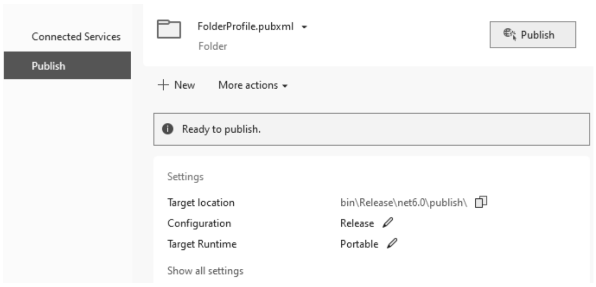
ClickOnce
Publish your application with ClickOnce

Docker Container Registry
Publish your application to any supported Container Registry that works with Docker images

Folder
Publish your application to a local folder or file share

Import Profile
Import your publish settings to deploy your app

Each of these options has its place, but for now, the simplest option is **Folder**. Select it and press **Next**. At this point, you will be asked whether you want to publish to a Folder or with ClickOnce. ClickOnce is a streamlined installer tool worth exploring another day. Pick **Folder** again to advance to the screen where you choose a directory. The default location is usually good enough, but pick any spot that works for you. That is enough to get the profile added to the profile list, but we still have more work to do.



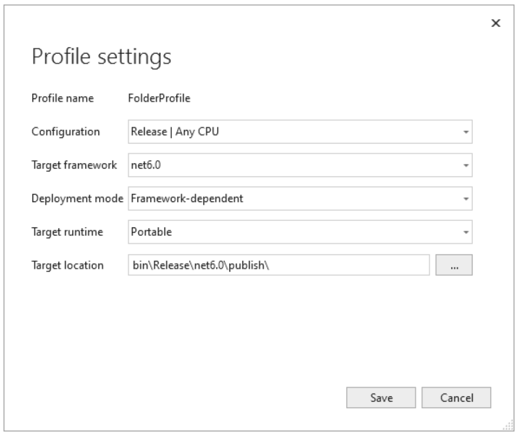
Ready to publish.

Settings

Target location	bin\Release\net6.0\publish\
Configuration	Release
Target Runtime	Portable

I usually start by renaming (**More Actions** > **Rename**) because FolderProfile is not very descriptive.

The interesting settings come when you hit the **Show all settings** button at the bottom, which opens the following dialog:


Profile settings

Profile name	FolderProfile
Configuration	Release | Any CPU
Target framework	net6.0
Deployment mode	Framework-dependent
Target runtime	Portable
Target location	bin\Release\net6.0\publish\

Each of these settings has a significant impact on the final product.

Configuration: The first setting allows you to choose which build configuration to use. The **Release | Any CPU** configuration is the default, and most of the time, that's what you want.

Target framework: This lets you pick which version of .NET you want to publish for. The default is usually correct, but you can choose any of the other frameworks you have installed.

Deployment mode: This is where things get interesting. There are two deployment modes. *Framework-dependent* deployment means that it expects the target version of .NET will be installed separately and already available on the machine it will run on. *Self-contained* will bundle the desired .NET runtime with your executable. If you know the target computer already has the correct version of .NET installed, framework-dependent is usually better because your artifacts will be smaller in size. Otherwise, you will want to use self-contained to include the runtime with your published artifacts.

If you use self-contained, you will have to pick an operating system and architecture, making separate publish profiles for each place you want to run on. Each publish profile will include the correct version of the runtime for running on the target computer.

Target runtime: This lets you determine the specific operating system and architecture you want to publish for. The default is **Portable**, though that is only an option if you use a framework-dependent deployment.

If you publish a framework-dependent deployment with a portable architecture, the result will be a *.dll* file containing your program. You launch a program like this with a command like `dotnet Program.dll`.

Alternatively, you can pick a specific runtime. If you use self-contained, this is a requirement. Aside from **Portable**, you will have options like **win-x86**, **win-x64**, **osx-x64**, and **linux-x64**. When you pick a non-portable runtime, a copy of the correct runtime will be included in the published artifacts. You can't run a Linux one on Windows or Windows on Linux, so this ties the published artifacts to a specific operating system and architecture. Make multiple profiles if you want to support multiple targets.

As long as you don't pick the Portable runtime, you will get several additional checkboxes under the **File publish options** group at the bottom, each of which has interesting impacts on its output.

Produce single file: Checking this box combines all of the outputs into a single file when publishing. And that means everything: your compiled code, the runtime, and other dependencies. It gives you a single file to share with users, which is convenient. There are ways to exclude specific files from the big bundle, so you have some control if you need it.

Enable ReadyToRun compilation: Checking this box will run the JIT compiler and produce hardware-ready instructions before you even deploy it. It takes the load off the JIT compiler as the program runs and can speed up launch times. ReadyToRun compilation results in a larger deployment size because the original CIL instructions are still included to support certain .NET features (like reflection).

Trim unused code: The full runtime, including the Base Class Library, is quite large. If you only use a small slice of it, much of this is waste. Trimming unused code will cut out the unused elements and reduce the file size. Be warned: there are ways to use code in the BCL that this feature can't detect, such as reflection. If you're using these tools, this option may cut out more than you intended. This option is available only with self-contained deployments.

.pubxml Files

Publish profiles are saved in the Properties/PublishProfiles folder with a *.pubxml* extension. You can edit these in a text editor, but it is usually safer to edit the profile in Visual Studio.

Publishing with a Profile

Once you have one or more profiles set up, it is easy to publish within Visual Studio by pushing the **Publish** button on the same screen you use to edit your publish profiles. Pushing this button kicks of the publishing process and places all of your compiled artifacts in the directory you chose.

After Publishing

Publishing assembles the output into the desired format but does not deploy the program. Deployment doesn't have to be complicated. You could put the artifacts in a *.zip* file or another archive format and then email the file or place it on a website for people to download. That is often sufficient for simple console applications like those in this book. For other app models, publishing may only be the first step.

Challenge **Altar of Publication** **100 XP**

To acquire the Medallion of Publishing, you must place a published program on the Altar of Publication.

Objectives:

- Select a program of yours for publication. This can be anything from Hello World to the most complex program you have made.
- Create a new publish profile. Choose appropriate settings based on how you want to publish your program.
- Publish your program.
- Package the output (for example, into a *.zip* file).
- Move the program to the target computer.
- Run the program successfully.
- **Note:** You will learn much by actually moving this to another computer. If you only have one, send it to a friend by email or the Internet. But if all of this fails, you can call the challenge done anyway.