Level 15

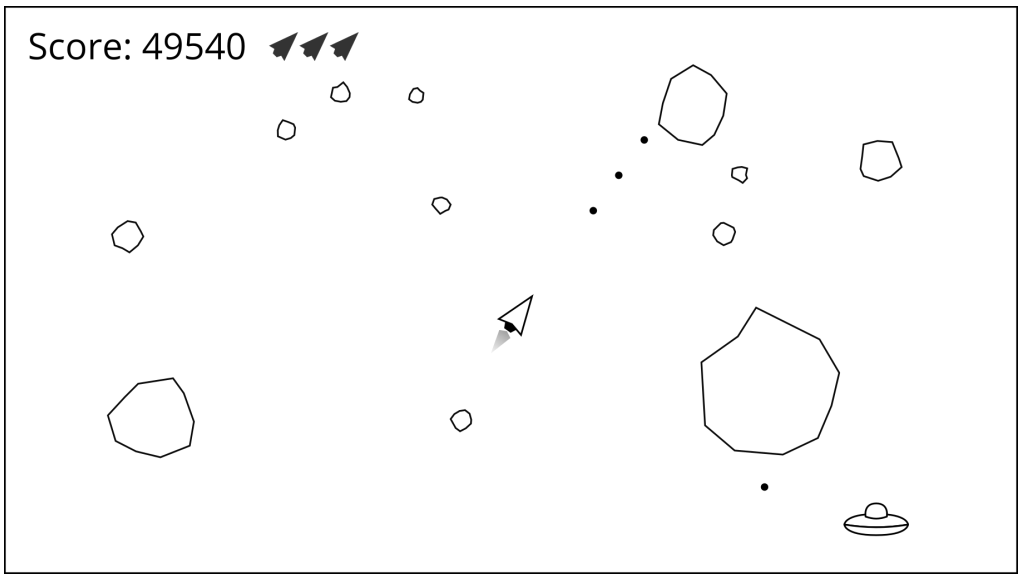
Object-Oriented Concepts

Speedrun

- Object-oriented programming allows you to separate large programs into individual components called objects, each responsible for a small slice of the overall program.
- Objects belong to a class, which defines a category of things with the same structure and capabilities.
- Building custom types is a powerful tool for building large programs.

Object-Oriented Concepts

In Part 2, we turn our attention from C# programming basics to a pair of problems with the same solution. Those two problems are shown in the picture below:



How do we go about building a program as complex as the game *Asteroids*, shown above? How do we take the raw ingredients we know and wield them to create something that spans hundreds or thousands of variables across thousands of methods?

Second, how do we represent complex concepts such as the ship, bullet, and asteroid shown above? What about concepts like the season of the year, dates, and scores on a high scores table?

The solution to both of these problems in C# is *object-oriented programming*.

The basic concept of object-oriented programming is that instead of putting all of our code into a single ever-growing blob of code, we split our program into multiple components called *objects*. Each object has a single responsibility (or perhaps a set of closely related responsibilities), and the objects work together to solve the overall problem.

Each object in the system performs its job in coordination with the other objects.

Objects can be created while the program runs. Objects that are no longer needed can be removed from the system.

Each object contains a set of methods and variables. Its variables store its data while its methods allow other objects to make requests of it. Most objects have a few of each. Some objects only need to represent the state something is in and contain only variables. Some do not need to remember any state or data and have only methods.

An object can coordinate with another object by calling one of its methods.

This paradigm is not unique to programming. For example, businesses and team projects work in the same way. Large challenges are too big for a single person, so the overall task is split among many people. They each fulfill their part of the larger system and can make requests and get information from others.

In C#, every object belongs to a specific *class* or *type*. An object's class determines the object's "shape." All objects of the same class have the same data elements and methods. You can think of classes as categories; everything in a class is similar in nature and structure.

Many different objects can belong to the same class. You can interact with objects of the same class in the same way, but each object is independent of the others. Objects of a particular class are often called *instances* of the class.

Many classes of objects already exist as a part of .NET. The **`string`** type is a class, for example. But C# allows us to define new classes as well.

This ability to define new types of objects gives us some hints about how we can solve our second problem: representing things that can't be represented well with any of the 14 simple types we learned about in Level 6. If one of the existing types isn't a good fit for something we need to represent, we can use these built-in types as building blocks to craft new tailor-made types to represent these more complex concepts.

For example, we could use three **`int`**s to represent a date on the calendar (day, month, and year), all bundled into a new class for representing dates. Or we could define a new class for representing asteroids, with their position, rotation, and speed, and give them methods for doing things asteroids do, like updating position over time.

C# has many ways to define new types from built-in ones, including enumerations, tuples, structs, and classes, with classes being the most sophisticated. Part 2 focuses on defining new types and especially on defining new classes. We'll see how to make new objects and get multiple objects working together to solve larger programs.

As we will see in the coming levels, when we encounter new concepts and ideas that don't fit nicely into the basic types we already know, we craft new types and classes to represent these

more complex concepts. Using objects, we will be able to build programs to solve more complex problems.

After defining a new type, we will be able to work with it as a cohesive new, reusable element—a new type that other variables can use. If we build a new **`Ship`** type, we will be able to make variables of the **`Ship`** type elsewhere in our program without recreating the logic and data that a ship encapsulates every time.

This concept will become an essential rule in C# programming: **Use the right type for everything you create. If the right type doesn't exist, create it first.**

Knowledge Check Objects 25 XP

Check your knowledge with the following questions:

1. What two things does an object bundle together?
2. **True/False.** C# lets you define new types of objects.

Answers: (1) data and operations on that data (methods). **(2)** True.

Level 16

Enumerations

Speedrun

- An enumeration is a custom type that lists the set of allowed values: `enum Season { Winter, Spring, Summer, Fall }`
- Define your enumerations after your main method and other methods or in a separate file.
- After defining an enumeration, you can use it for a variable's type: `Season now = Season.Winter;`

In C#, types matter. You don't use a **`string`** when working with numbers, and you don't use an **`int`** when working with text. What do we do when we encounter something that doesn't fit nicely into one of our pre-existing types? For example, what if we need to represent the seasons of the year (winter, spring, summer, fall)?

Using only data types that we're already familiar with, we have two choices: an integer type like **`int`** or a **`string`**. With an **`int`**, we could assign a number to each season:

```
int current = 2; // Summer
```

And:

```
if (current == 3) Console.WriteLine("Winter is coming.");
```

This approach can work but has two problems. First, it's hard to remember which season is which. Did we start with winter or spring? Do we start counting at 0 or 1? Only with the comment does it become clear. The second problem is that nothing prevents us from using weird numbers. Somebody could make the current season -14 or 2 million.

What if we used strings? We could use the text **`"Summer"`** to represent summer:

```
string current = "Summer";
```

And:

```
if (current == "Fall") Console.WriteLine("Winter is coming.");
```

This approach has similar problems. While the text **`"Fall"`** is far less likely to be misinterpreted, **`"Fall"`**, **`"fall"`**, **`"Faall"`**, and **`"Autumn"`** are not the same string. And nothing prevents us from doing something like **`current = "Monday";`**.

C# provides a better solution to this problem: defining a new type called an enumeration.

Enumeration Basics

An *enumeration* or an *enumerated type* is a type whose choices are one of a small list of possible options. The verb *enumerate* means "to list off things, one by one," hence the name. We can define new enumerations in our code to represent concepts of this nature.

Enumerations only work when you have a relatively small set (a few, tens, or maybe hundreds) of choices, especially when you can make an exhaustive list, not leaving anything out. For example, the Boolean values **`true`** and **`false`** would be an excellent enumeration if they were not already part of the **`bool`** type. With only four choices, the year's seasons are also a great candidate for an enumeration.

Defining an Enumeration

Before we can use an enumeration, we have to define it. New type definitions, including enumerations, must come after our main method and the methods it owns (or in a separate file, as we will do later). However, when we create multiple new types, their relative order does not matter.

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

// <-- Add new enumerations here, at the end of your file.
```

The following defines a new enumeration to represent seasons:

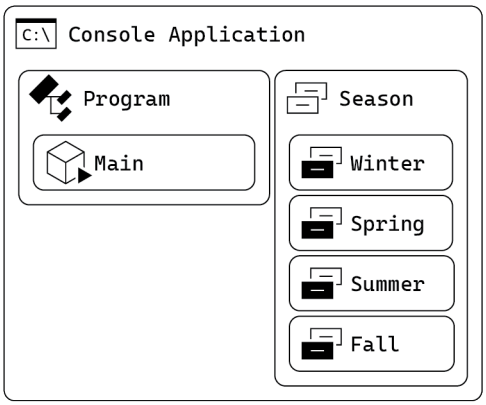
```
enum Season { Winter, Spring, Summer, Fall }
```

To define a new enumeration, you start with the **`enum`** keyword, followed by the enumeration's name (**`Season`**). A set of curly braces contains the options for the enumeration, separated by commas. In C#, it is common to use UpperCamelCase for type names (including enumeration names) and enumeration members. The above code used **`Season`** instead of **`season`** or **`SEASON`**, and **`Winter`** instead of **`WINTER`** or **`winter`** for that reason. Of course, the choice is yours, but I recommend giving this standard convention a try.

Once placed in the rest of the code, the entire file might look like this:

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

enum Season { Winter, Spring, Summer, Fall } // New types MUST go after other
                                             // code (or in another file).
```

Unlike methods, type definitions like this do *not* live inside your main method. The code map looks like this instead:



Once your file encounters a type definition like this, it marks it as the end of your main method, and no further statements can come afterward. But you can add as many types to the bottom of your file as you want. Some people prefer putting new type definitions, like an enumeration, into separate *.cs* files. We'll cover that in Level 33, but feel free to jump ahead and read that section if you think you'd prefer separate files.

Whitespace does not matter, so the following style is also typical:

```
enum Season
{
    Winter,
    Spring,
    Summer,
    Fall
}
```

The first item you list will be the enumeration's default value, so choose it wisely.

Using an Enumeration

With our **`Season`** enumeration defined, we can use it like any other type. For example, we can declare a variable whose type is **`Season`**:

```
Season current;
```

The compiler can now help us enforce that only legitimate seasons are assigned to this variable. You can pick a specific value like this:

```
Season current = Season.Summer;
```

We access a specific enumeration value through the enumeration type name and the dot operator. This is a bit more complicated than literals like **`2`** or **`"Summer"`**, had we just used **`int`**s or **`string`**s, but it is not bad.

Enumerations have much in common with integers. For example, we can use the **`==`** operator to check for equality:

```
Season current = Season.Summer;

if (current == Season.Summer || current == Season.Winter)
    Console.WriteLine("Happy solstice!");
else
    Console.WriteLine("Happy equinox!");
```

```
enum Season { Winter, Spring, Summer, Fall } // New types MUST go after other
                                             // code (or in another file).
```

Revisiting `ConsoleColor`

In the past, we have used the **`ConsoleColor`** type like this:

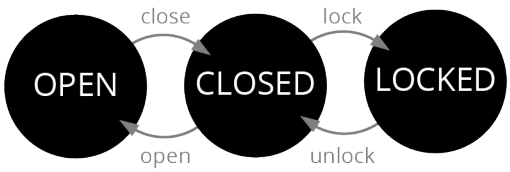
```
Console.BackgroundColor = ConsoleColor.Yellow;
```

That code should have new meaning now. **`ConsoleColor`** is an enumeration! Somewhere out there is code like **`enum ConsoleColor { Black, Yellow, Red, ... }`**. Equipped with the knowledge of enumerations, we could have written that ourselves!

Challenge — Simula's Test — 100 XP

As you move through the village of Enumerant, you notice a short, cloaked figure following you. Not being one to enjoy a mysterious figure tailing you, you seize a moment to confront the figure. "Don't be alarmed!" she says. "I am Simula. They are saying you're a Programmer. Is this true?" You answer in the affirmative, and Simula's eyes widen. "If you are truly a Programmer, you will be able to help me. Follow me." She leads you to a backstreet and into a dimly lit hovel. She hands you a small, locked chest. "We haven't seen any Programmers in these lands in a long time. And especially not ones that can craft types. If you are a True Programmer, you will want what is in that chest. And if you are a True Programmer, I will gladly give it to you to aid you in your quest."

The chest is a small box you can hold in your hand. The lid can be open, closed (but unlocked), or locked. You'd normally be able to go between these states, opening, closing, locking, and unlocking the box, but the box is broken. You need to create a program with an enumeration to recreate this locking mechanism. The image below shows how you can move between the three states:



Nothing happens if you attempt an impossible action in the current state, like opening a locked box.

The program below shows what using this might look like:

```
The chest is locked. What do you want to do? unlock
The chest is unlocked. What do you want to do? open
The chest is open. What do you want to do? close
The chest is unlocked. What do you want to do?
```

Objectives:

- Define an enumeration for the state of the chest.
- Make a variable whose type is this new enumeration.
- Write code to allow you to manipulate the chest with the **`lock`**, **`unlock`**, **`open`**, and **`close`** commands, but ensure that you don't transition between states that don't support it.
- Loop forever, asking for the next command.

UNDERLYING TYPES

The deep dark secret of enumerations is that they are integers at heart, though the compiler will ensure you don't accidentally misuse them. Each enumeration has an *underlying type*, which is the integer type that it builds upon. The default underlying type is **`int`**, but you could change that:

```
enum Season : byte { Winter, Spring, Summer, Fall }
```

It is usually not worth the trouble to change this, but it is worth considering if memory is tight.

Because enumerations are based on integers, there are some other tricks you may find useful. Each enumeration member is assigned an **`int`** value. By default, these are given in the order they appear in the definition, starting with **`0`**. So above, **`Winter`** is **`0`**, **`Spring`** is **`1`**, etc. If you want, you can assign custom numbers:

```
enum Season { Winter = 3, Spring = 6, Summer = 9, Fall = 12 }
```

Any enumeration member without an assigned number is automatically given the one after the member before it. So below, **`Winter`** is **`1`**, **`Spring`** is **`2`**, **`Summer`** is **`3`**, and **`Fall`** is **`4`**:

```
enum Season { Winter = 1, Spring, Summer, Fall }
```

The default value for an enumeration is whichever one is assigned the number 0. That remains true even if nothing is assigned 0, which means the default value may not even be a legal choice! In that case, consider adding something like **`Unknown = 0`** so you can still refer to the default value by name.

You can also cast between **`int`**s and enumerations:

```
int number = (int)Season.Fall;
Season now = (Season)2;
```

Use this cautiously. It can result in using a number that is not a valid enumeration option:

```
Season another = (Season)822; // Not a valid season!
```

Level 17

Tuples

Speedrun

- Tuples combine multiple elements into a single bundle: `(double, double) point = (2, 4);`
- You can give (ephemeral) names to tuple elements, which can be used later: `(double x, double y) point = (2, 4);`
- Tuples can be used like any other type, including variable and return types.
- Deconstruction unpacks tuples into multiple variables: `(double x, double y) = MakePoint();`
- Two tuple values are equal if their corresponding items are all equal.

The next tool we will acquire in our arsenal combines many variables into a single bundle: a tuple. Before we go too far, I need to point out that tuples have their place, but we will soon learn better tools for most situations. Most C# programmers only use tuples occasionally.

To understand where we can use tuples, let's consider the problem they solve, illustrated by the picture below:

This picture is roughly what the original *Tetris* high score table looks like. How could we represent these scores in our program? These scores are more than just a single `int` value. Each score has the player's name, points, and the level they reached in getting the score.

We could imagine making three variables along the lines of **`string name`**, **`int points`**, and **`int level`** for just a single score. But to make the full table, we need three of each. We could do this with arrays:

```
string[] names = new string[3] { "R2-D2", "C-3PO", "GONK" };
int[] points   = new int[3]    {   12420,    8543,     -1 };
int[] level    = new int[3]    {      15,       9,      1 };
```

But this feels like we've organized our data sideways. Instead of putting R2-D2 with his score and level, we have put all names, points, and levels together.

In this case, a score feels like its own concept or idea. And as we learned in Level 15, we should make a new type to capture the idea when this happens. We need some way to represent an entire score's information—name, point total, and level—in a bundle. When multiple data elements are combined like this, it is sometimes referred to as a *composite type* because the larger thing is *composed of* the smaller pieces. Or you could say that we use *composition* to build the larger element.

THE BASICS OF TUPLES

In C#, the simplest tool for creating composite types is called a *tuple* (pronounced "TOO-ples" or "TUP-ples"). A tuple allows us to combine multiple pieces into a single element. The name comes from the math world to generalize the naming pattern *double*, *triple*, *quadruple*, *quintuple*, etc. These are also sometimes referred to by the number of items in them: a 2-tuple if it has two things, an 8-tuple if it has eight things, etc.

Forming a new tuple value is as simple as taking the pieces you need and placing them in parentheses, separated by commas:

```
(string, int, int) score = ("R2-D2", 12420, 15);
```

The variable type is formed similarly, listing the types in parentheses, separated by commas. That leads to a long type name, and while I have been avoiding **`var`** for clarity, this is a good example of why some people prefer **`var`**:

```
var score = ("R2-D2", 12420, 15);
```

The type for **`score`** is a 3-tuple composed of a **`string`**, an **`int`**, and an **`int`**.

You can access the items inside of the tuple like so:

```
Console.WriteLine($"Name:{score.Item1} Level:{score.Item3} Score:{score.Item2}");
```

These names leave a lot to be desired. Was it **`Item2`** or **`Item3`** that contained the point total? It is easy to get them mixed up, and it gets worse with tuples with many items. We will soon see ways to attach alternative names to the items in a tuple, but behind the scenes, the names really are **`Item1`**, **`Item2`**, and **`Item3`**.

The type of a tuple is determined by the type and order of the parts of the tuple. That means you can do something like this:

```
(string, int, int) score1 = ("R2-D2", 12420, 15);
(string, int, int) score2 = score1; // An exact match works.
```

But you cannot do either of these:

```
(string, int) partialScore = score1;        // Not the same number of items.
(int, int, string) mixedUpScore = score1;   // Items in a different order.
```

Tuples are value types, like **int**, **bool**, and **double**. That means they store their data inside them. Assigning one variable to another will copy all the data from all of the items in the process. That is made a bit more complicated because tuples are composite types. If a tuple has parts that are value types themselves, those bytes will get copied. But if an item is a reference type, then the reference is copied.

TUPLE ELEMENT NAMES

The names of the items in a tuple are **Item1**, **Item2**, etc. Behind the scenes, that is precisely how they work. However, the compiler lets you *pretend* that the items have alternative names. Doing so can lead to much more readable code.

If you don't use **var**, you can assign names to each item in the tuple like so:

```
(string Name, int Points, int Level) score = ("R2-D2", 12420, 15);
Console.WriteLine($"Name:{score.Name} Level:{score.Level} Score:{score.Points}");
```

Placing names next to the types when the variable is declared, you will be able to refer to those names later, as shown in the second line.

You are not required to give a name to every tuple member. Any unnamed item will keep its original **ItemN** name:

```
(string Name, int, int) score = ("R2-D2", 12420, 15);
Console.WriteLine($"Name:{score.Name} Level:{score.Item3} Score:{score.Item2}");
```

If you use **var**, you lose your chance to give the items a name in this manner. But you are not out of luck. You can also apply names to a tuple when the tuple is formed:

```
var score = (Name: "R2-D2", Points: 12420, Level: 15);
Console.WriteLine($"Name:{score.Name} Level:{score.Level} Score:{score.Points}");
```

When you use **var** in this way, not only are the tuple's constituent types inferred, but the names will be as well.

However, if you do not use **var**, then the names will not be inferred, and any name supplied when you declared the variable would be used:

```
(string, int P, int L) score = (Name: "R2-D2", Points: 12420, Level: 15);
Console.WriteLine($"Name:{score.Item1} Level:{score.L} Score:{score.P}");
```

With the above code, the names are **Item1**, **P**, and **L**, not **Name**, **Points**, and **Level**.

These examples help illustrate that even though adding names can lead to clearer code, the names are fluid and are not a part of the tuple itself. For tuples, names are only cosmetic.

TUPLES AND METHODS

While tuple types are more complicated, they are just another type for all practical purposes. For example, you can use them as parameter types or return values. We can take the code we have been working with and turn it into a method for displaying scores, passing in a score as a tuple:

```
void DisplayScore((string Name, int Points, int Level) score)
{
    Console.WriteLine(
            $"Name:{score.Name} Level:{score.Level} Score:{score.Points}");
}
```

Alternatively, we could have left out the tuple element names and just used **`Item1`**, **`Item2`**, and **`Item3`** in the method itself. Parameters cannot use **`var`**, so we are obligated to list the tuple item types in this case.

The syntax here is trickier because tuples and the parameters of a method both use parentheses and commas. You will want to pay careful attention when using it this way.

A parameter whose type is a tuple is just another parameter, and you can mix and match tuple parameters with non-tuple (normal) parameters as needed.

The same is true of return types. You can return a tuple from a method by placing its constituent parts in parentheses (names optional) in the spot where we list the return type:

```
(string Name, int Points, int Level) GetScore() => ("R2-D2", 12420, 15);
```

A tuple's types and names can be inferred from a called method's return type, just like when we created the new value inline:

```
var score = GetScore();
Console.WriteLine($"Name:{score.Name} Level:{score.Level} Score:{score.Points}");
```

But the names provided by your return value do not need to match those of your variable. This is shown below, where everything is using different names:

```
(string One, int Two, int Three) score = GetScore();
DisplayScore(score);

(string N, int P, int L) GetScore() => ("R2-D2", 12420, 15);

void DisplayScore((string Name, int Points, int Level) score)
{
    Console.WriteLine(
                $"Name:{score.Name} Level:{score.Level} Score:{score.Points}");
}
```

This illustrates more clearly that names are ephemeral and not a part of the tuple.

MORE TUPLE EXAMPLES

Let's look at a few more examples of tuples before moving on.

This tuple represents a point in two-dimensional space:

```
(double X, double Y) point = (2.0, 4.0);
```

Think of how nice it could be to combine these two coordinates into a single thing and pass it around in your code if you were making a game in a 2D world.

Or, if we have a grid-based world, what about using a tuple with elements for the grid square's type and location? We could define the tile's type as an enumeration like this:

```
enum TileType { Grass, Water, Rock }
```

We can place that into a tuple with a row and a column:

```
var tile = (Row: 2, Column: 4, Type: TileType.Grass);
```

And here is a tuple with 16 elements to show a much bigger tuple, representing a 4×4 matrix—something often used in games:

```
var matrix = (M11: 1, M12: 0, M13: 0, M14: 0,
              M21: 0, M22: 1, M23: 0, M24: 0,
              M31: 0, M32: 0, M33: 1, M34: 0,
              M41: 0, M42: 0, M43: 0, M44: 1);
```

(Perhaps an array would be better for that last one?)

And finally, let's look at an example that creates and returns an array of **`(string, int, int)`** tuples to create the full scoreboard we introduced at the beginning of this level:

```
(string Name, int Points, int Level)[] CreateHighScores()
{
    return new (string, int, int)[3]
    {
        ("R2-D2", 12420, 15),
        ("C-3PO",  8543,  9),
        ("GONK",     -1,  1),
    };
}
```

The above code creates a fixed list of scores, but in a real-world situation, we'd probably store these in a file and load them from there (Level 39).

DECONSTRUCTING TUPLES

We have seen many examples of creating tuples. Let's look at the opposite. Suppose you have the following tuple:

```
var score = (Name: "R2-D2", Points: 12420, Level: 15);
```

The simplest way to grab data out of a tuple is just to reference the item by name:

```
string playerName = score.Name;
```

When you only need a single item from the tuple, this is a good way to do it.

But there is a way to take all of the parts of a tuple and place them each into separate variables all at once. This is called *deconstruction* or *unpacking*. It is done by listing each of the variables to store the deconstructed tuple in parentheses:

```
string name;
int points;
int level;

(name, points, level) = score;
Console.WriteLine($"{name} reached level {level} with {points} points.");
```

The highlighted line copies each item in the tuple to their respective variables.

You can declare new variables at the same time, so we could also have written the above code like this:

```
(string name, int points, int level) = score;
```

That starts to look precariously close to declaring a new tuple variable with named items. The difference is that this version does not provide a name after the parentheses to refer to the entire tuple.

Tuple deconstruction has many uses, but a clever usage is swapping the contents of two variables:

```
double x = 4;
double y = 2;
(x, y) = (y, x);
```

The two variables' contents are copied over to a new tuple and then copied back to **x** and **y**. The result is **x** and **y** have swapped values with only a single line.

Ignoring Elements with Discards

Tuple deconstruction demands that the variables on the left match the tuple in count and types. Sometimes, you only care about some of the values. Rather than make a variable called **junk** or **unused**, you can use a discard variable using a simple underscore, and no type:

```
(string name, int points, _) = score;
```

The **_** is a discard variable. The compiler will invent a name for it behind the scenes so the code can work, but it won't clutter up the code with useless names and leads to more readable code. Wins all around.

TUPLES AND EQUALITY

Tuples are value types and thus, use value semantics when checking for equality. Two tuple values are considered equal if they have the same number of items, the corresponding items are the same types, and if each item is equal to the corresponding item in the other tuple. That last item is a little tricky because if some part of a tuple is a reference type, then the references (and not the data) will be checked for equality. The following will display **True** and then **False**:

```
(int, int) a = (1, 2);
(int, int) b = (1, 2);

Console.WriteLine(a == b);
Console.WriteLine(a != b);
```

There is one potential surprise to tuple equality. Will **a** and **b** below be equal or not equal?

```
var a = (X: 2, Y: 4);
var b = (U: 2, V: 4);
Console.WriteLine(a == b);
```

The only difference is the names given to the tuple elements. Do the names of the tuple elements matter? Since names are not officially part of the tuple, **a** and **b** above are equal despite the name differences.

Challenge Simula's Soup 100 XP

Simula is impressed with how you reconstructed the box with an enumeration. When the box opened, you saw a glowing emerald gem inside. You don't know what it is, but it seems important. Also in the box were three vials of powder labeled HOT, SALTY, and SWEET.

"Finally! I can make soup again!" Simula says. She casually tosses the small glowing gem to you but is wholly focused on the powders. "You stick around and help me make soup with your programming skills, and I'll tell you what that gem does."

She pulls out a cookpot, knocks the clutter off the table with a quick sweep of her arm, and begins cooking. She says, "I'm the best soup maker in town, and you're in for a treat. I've got recipes for soup, stew, and gumbo. I've got mushrooms, chicken, carrots, and potatoes for ingredients. And thanks to you getting that box open, I've got seasonings again! Spicy, salty, and sweet seasoning. Pick a recipe, an ingredient, and a seasoning, and I'll make it. Use your programming skills to help us track what we make."

Objectives:

- Define enumerations for the three variations on food: type (soup, stew, gumbo), main ingredient (mushrooms, chicken, carrots, potatoes), and seasoning (spicy, salty, sweet).
- Make a tuple variable to represent a soup composed of the three above enumeration types.
- Let the user pick a type, main ingredient, and seasoning from the allowed choices and fill the tuple with the results. **Hint:** You could give the user a menu to pick from or simply compare the user's text input against specific strings to determine which enumeration value represents their choice.
- When done, display the contents of the soup tuple variable in a format like "Sweet Chicken Gumbo." **Hint:** You don't need to convert the enumeration value back to a string. Simply displaying an enumeration value with **`Write`** or **`WriteLine`** will display the name of the enumeration value.)

Narrative The Fountain of Objects

As you eat soup with Simula, she explains that she is the Caretaker of the Heart of Object-Oriented Programming—the glowing green gem in the box. For thousands of clock cycles, she has held onto it, hoping that someday, a Programmer who understood object-oriented programming would appear to restore the Fountain of Objects, destroyed by The Uncoded One, back to what it once was: the lifeblood of the entire island.

She tells you that to do this, you must gather the five keys of Object-Oriented Programming and make your way to the Fountain of Objects, whose location is secret. She tells you you can discover its location if you visit the Catacombs of the Class and marks that location on your map.

You leave Simula's hovel behind and begin the quest to restore the Fountain of Objects to what it once was. Your next destination: the Catacombs of the Class!

Level 18

Classes

Speedrun

- Classes are the most powerful way to define new types.
- A class bundles together data (fields) and operations on that data (methods): `class Score { public int points; public int level; public void Method() { } }`
- Constructors define how new instances are created: `public Score(int p) { points = p; }`
- Classes are reference types.

We got our first taste of making and using custom types with enumerations. We got our first taste of composite types with tuples. With the appetizers out of the way, it is time to dive into the main course: classes. Classes are the king of the object-oriented world. We'll revisit representing a score once again, this time using classes to solve the problem.

Let's start by giving official definitions for the concepts of objects, classes, and instances.

An *object* is a thing in your software, responsible for a slice of the entire program, containing data and methods, which define what information the object must remember and the capabilities it can perform when requested.

An object-oriented program typically has many objects, each performing its own job in the system. Some objects know about other objects and work with them to get their jobs done by invoking others' methods. We have already been doing this in the programs we have created. Our main method lives in an object and asks the **`Console`**, **`Convert`**, and **`Math`** objects to perform tasks from the ones they are built to do. (Though we will soon see that **`Console`**, **`Convert`**, and **`Math`** fit into a different category than most objects.)

A big part of programming in an object-oriented world is deciding how to split the program into objects. Which responsibilities should each have? What data and methods does the object need to fulfill those responsibilities? Which other objects does it need to work with? These questions are at the heart of *software design*, or more specifically, *object-oriented design*. We will get into this topic later in this book, but it is also a topic that takes months and years of study and practice to get good at. On the bright side, software is soft—malleable. If you try something and later decide that another way is better, you can change the code.

In some programming languages, objects are flexible. Variables and methods can be added and stripped from an object over time. This scheme is great for tiny programs because there is low formality and high flexibility. But you can never be sure that an object has a particular piece of data or is capable of performing a specific method. As your programs get larger in this scheme, this problem grows out of control.

In C#, types matter. Rather than morphing over their lifetime, C# objects are categorized into *classes*. By defining a class, you establish the variables and methods of any object belonging to the class. You can think of a class definition as a blueprint or pattern for objects that belong to the class.

Programmers will also refer to objects that fit into a class as an *instance* of that class. For example, if we define a **`Score`** class, an object that belongs to the **`Score`** class might be called an instance of the **`Score`** class or a **`Score`** instance. The words "object" and "instance" are almost synonyms, though "object" is used more often when the specific class matters less, while "instance" is typically used in conjunction with a type name.

Defining a class also defines a new type that you can use for variables. Classes are reference types, putting them in the same bucket as strings and arrays. Variables whose type is a class hold only a reference; the object's data lives somewhere on the heap. In this book, I sometimes use the word "value" to indicate the contents of a value type and "object" to indicate the contents of a reference type, though these terms are often blurred together in the programming world.

DEFINING A NEW CLASS

Before we can use a class, we must first define it. Many C# programmers place each class in a separate file. Indeed, as your programs grow big enough to have 10 or 100 classes defined, you will not want to keep it all in a single file. We will see how to split your program across many files later (Level 33, though you can probably figure it out on your own). For now, you can place them in the same place we have placed enumerations, at the bottom of our main file, after all other statements and methods.

Defining a new class is done with the **`class`** keyword, followed by the class's name, followed by a set of curly braces. Names are usually capitalized with UpperCamelCase, just like enumerations and methods. Inside the class's curly braces, we can place the variables and methods that the class will need to do its job.

Using the *Tetris* score table example from the previous level, we know we need three variables: a name, a point total, and the level the player reached. A simple **`Score`** class looks like this:

```
// <-- Your main method goes here.

class Score
{
    public string name;
    public int points;
    public int level;
}

// <-- Other classes and enumerations can go here.
```

These variables are not the same thing as local variables or parameters. They are another category of variables called *fields* or *instance variables*. Local variables and parameters belong

to a method and come and go as the method is called and ends. Fields are variables created inside the object's memory on the heap. They live for as long as the object lives and are a part of the object itself.

In Level 19, we will look at what that **`public`** does, but for now, we will just blindly apply that to the fields we make.

Unlike a tuple, we can also add methods to a class. For example, the method below indicates whether the score earned a star, defined by averaging at least 1000 points per level:

```
class Score
{
    public string name;
    public int points;
    public int level;

    public bool EarnedStar() => (points / level) > 1000;
}
```

That **`EarnedStar`** method is like most methods we have seen in the past, but with two notable differences. The first is that it also has a **`public`** on it. Again, for now, we will just assume that's what you do for methods that belong to a class.

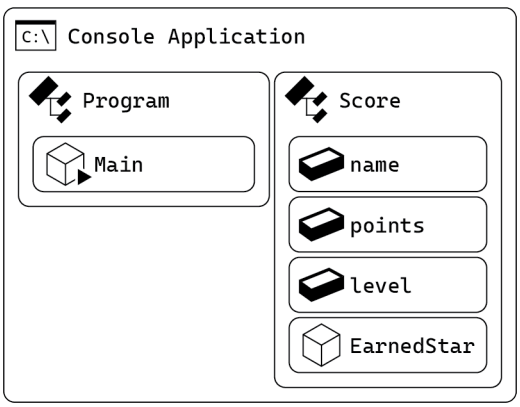
Perhaps the most notable part of that method is how it uses the **`points`** and **`level`** fields in its code. This lines up with what we've seen in the past about scope. Since **`EarnedStar`** lives in **`Score`**, this method will have access to its own local variables and parameters (though this method has neither) and also any variables defined in the class itself.

Classes give us a way to bundle together data and the operations on that data into a well-defined cohesive unit. This principle is called *encapsulation*. That principle is so important of an idea that we want to call it out formally and give it a name. It is the first of five object-oriented principles that form the foundation of object-oriented programming.

Object-Oriented Principle #1: Encapsulation—Combining data (fields) and the operations on that data (methods) into a well-defined unit (like a class).

Encapsulation is a crucial element in building objects that solve a slice of the overall problem, which lets us make larger programs.

Much like what we saw with enumerations, when we define classes, they do not live within the main method but are separate:



INSTANCES OF CLASSES

The code above defines the **Score** class. It describes how scores in our program will work. It provides the blueprint for all scores that come into existence as our program runs.

With a class defined, we can use it like any other type. We can declare a variable whose type is **Score**, for example, and then assign it a new instance:

```
Score best = new Score();

class Score
{
    public string name;
    public int points;
    public int level;

    public bool EarnedStar() => (points / level) > 1000;
}
```

Instances of a class are created with the **new** keyword. That **Score()** thing refers to a special method called a *constructor*, used to get new instances ready for use. We didn't define such a constructor in our **Score** class, but the compiler was nice enough to generate a default one for us. That is what is being used here. We will see how to define our own in a moment. The expression **new Score()** creates a new instance of the **Score** class, placing its data on the heap (it is a reference type, after all) and grabbing a reference to it. That reference is then stored in the **best** variable.

Now that our instance has been created, we can work with its fields and invoke its methods:

```
Score best = new Score();

best.name = "R2-D2";
best.points = 12420;
best.level = 15;

if (best.EarnedStar())
    Console.WriteLine("You earned a star!");
```

The middle section of that code assigns new values to each of the instance's fields. These fields belong to the instance, so we must access them through a reference to an instance, such as the one contained in **best**.

In the **if** statement's condition, the **EarnedStar** method is invoked. This is different from how we have invoked methods before. Here, too, we must access the method through an instance of the **Score** class, such as the one contained in the **best** variable. It is more like how we call **Console**'s and **Convert**'s methods. However, in those cases, we used the class name rather than using an instance. We'll sort out that particular difference in Level 21.

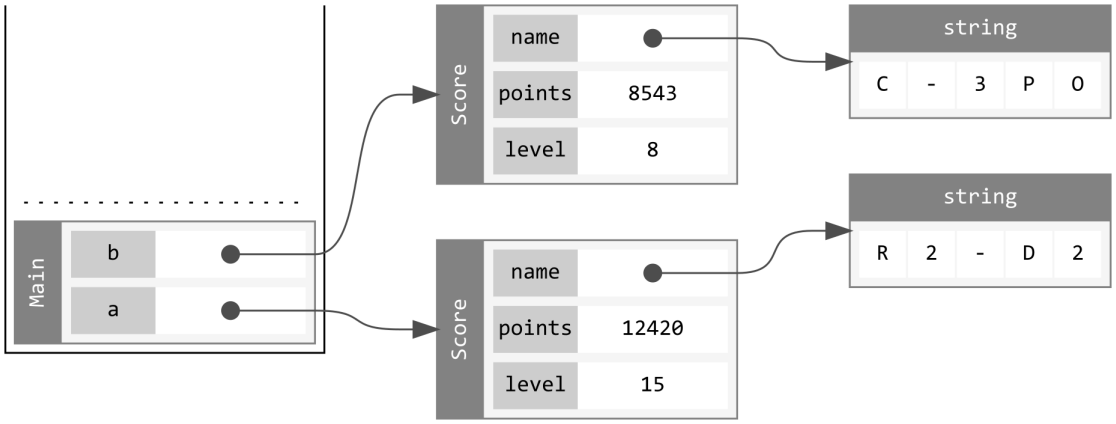
We can create more than one instance of a class. When we do this, each instance has its own data, independent of the other instances:

```
Score a = new Score();
a.name = "R2-D2";
a.points = 12420;
a.level = 15;

Score b = new Score();
b.name = "C-3PO";
```

```
b.points = 8543;
b.level = 8;

if (a.EarnedStar())
    Console.WriteLine($"{a.name} earned a star!");
if (b.EarnedStar())
    Console.WriteLine($"{b.name} earned a star!");
```

This code creates two **Score** instances and places a reference to each in the variables **a** and **b**. Because each instance has its own data, when we call **a.EarnedStar()**, it is making the determination based on **a**'s data, and for **b.EarnedStar()**, on **b**'s data.

If we look at the memory used in the program above, it looks like this after running:



CONSTRUCTORS

Creating a new object reserves space for the object's data on the heap. But it is also vital that new objects come into existence in legitimate starting states. *Constructors* are special methods that run when an object comes to life to ensure it begins life in a good state. The following adds a constructor to the **Score** class, giving each field a good starting value:

```
class Score
{
    public string name;
    public int points;
    public int level;

    public Score()
    {
        name = "Unknown";
        points = 0;
        level = 1;
    }

    public bool EarnedStar() => (points / level) > 1000;
}
```

Constructors are like other methods in most ways, but with two caveats. Constructors must use the same name as the class, and they cannot list a return type. Otherwise, constructors are essentially the same as any other method. Add **if** statements, loops, and call other methods as needed.

A constructor's job is to get new instances into a legitimate starting state. The specifics will vary from class to class, but assigning initial values to each field is common.

Default Constructors and Default Field Values

At this point, you may be wondering about the fact that we didn't define a constructor before but could still create new instances of the **`Score`** class. How did that work?

If you don't define any constructors, the compiler inserts one that looks like this:

```
public Score() { }
```

The constructor exists and can be used when creating new instances, but it doesn't do anything fancy. The purpose of a constructor is to put new instances of the class into a valid starting state. Yet this constructor doesn't initialize anything. What is the starting state of our fields in this case? Like we saw with arrays (Level 12), each field is initialized to the type's default value. This initialization is done by filling the object's memory with all 0 bits. In fact, anything allocated on the heap is initialized in this same way, which is why we get default values in both arrays and new objects. As we saw before, the **`int`** type's default value is the number **`0`**, and the **`string`** type's default value is the special value **`null`** (a lack of a value, and something we will cover in more depth in Level 22). Thus, a new **`Score`** instance would have had a **`null`** name (a lack of a name), with 0 points and a level of 0.

As soon as we add our own constructor to a class, the default one no longer auto-generates.

Constructors with Parameters

Constructors are allowed to have parameters, just like other methods. It is quite common for constructors to use parameters to let the outside world provide the initial values for some fields. The constructor below does this:

```
class Score
{
    public string name;
    public int points;
    public int level;

    public Score(string n, int p, int l)  // That's a lowercase 'L', not a 1.
    {
        name = n;
        points = p;
        level = l;
    }

    public bool EarnedStar() => (points / level) > 1000;
}
```

The names **`n`**, **`p`**, and **`l`** are not good variable names. Normally, I'd name them **`name`**, **`points`**, and **`level`**, but that causes a problem. Fields, local variables, and parameters are all accessible from inside a class's methods, including constructors. A local variable or parameter is allowed to have the same name as a field, but when they share names, weird things happen. We'll sort that out in a minute, but we'll use the names **`n`**, **`p`**, and **`l`** to avoid sharing names for now.

This constructor has three parameters, letting the calling code provide initial values for each field.

With this new constructor, we will need to change how we ask for a new **`Score`** instance, but with this change, we no longer need to initialize each field afterward.

```
Score score = new Score("R2-D2", 12420, 15);
```

Multiple Constructors

A class can define as many constructors as you need. Each of these must differ in number or types of parameters. The code below defines two constructors. The first one has no parameters (a *parameterless* constructor) and gives each field a reasonable starting value. The second constructor has three parameters to supply initial values for each field.

```
class Score
{
    public string name;
    public int points;
    public int level;

    public Score()
    {
        name = "Unknown";
        points = 0;
        level = 1;
    }

    public Score(string n, int p, int l)
    {
        name = n;
        points = p;
        level = l;
    }

    public bool EarnedStar() => (points / level) > 1000;
}
```

With two constructors, the outside world pick which constructor it wants to use:

```
Score a = new Score();
Score b = new Score("R2-D2", 12420, 15);
```

Initializing Fields Inline

Another way to initialize fields is by doing so inline, where they are declared, as shown below:

```
class Score
{
    public string name = "Unknown";
    public int points = 0;
    public int level = 1;

    public bool EarnedStar() => (points / level) > 1000;
}
```

These assignments happen after the memory is zeroed out but before any constructor code runs. These then become the default values for these fields. If these defaults are sufficient and no other initialization needs to happen, you can skip defining your own constructors. But any constructor can also override these defaults as needed:

```
class Score
{
    public string name = "Unknown";
    public int points;
    public int level = 1;

    public Score()
    {
        name = "Mystery";
    }

    public bool EarnedStar() => (points / level) > 1000;
}
```

The **`points`** field will take on the default **`int`** value of **`0`**. The **`level`** field will be assigned a value of **`1`** because of the field's initializer. **`name`** will first be assigned **`"Unknown"`** and subsequently updated with **`"Mystery"`**.

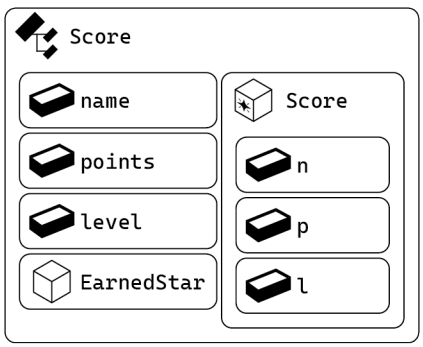
Like parameters, we cannot use **`var`** for a field's type. It must always be written out.

Name Hiding and the `this` Keyword

Let's get back to those bad single-letter variable names:

```
public Score(string n, int p, int l)
{
    name = n;
    points = p;
    level = l;
}
```

Our **`Score`** class, with this constructor, looks like this on a code map:



Within the **`Score`** constructor, we have access to the **`n`**/**`p`**/**`l`** set of variables and the **`name`**/**`points`**/**`level`** set. But those single-letter names are not great. Typically, I'd have given **`n`**, **`p`**, and **`l`** the names **`name`**, **`points`**, and **`level`**. In this case, doing so would use those names twice. For better or worse, C# allows this. But consider what happens when you do:

```
public Score(string name, int points, int level)
{
    name = name; // These will not do what you want!
    points = points;
    level = level;
}
```

On a line like **`name = name;`**, both usages of **`name`** refer to the element in the more narrow scope, which is the constructor parameter. This code takes a variable's content and assigns it back into that same variable. The class's **`name`** field is technically still in scope, but the parameter with the same name hides access to it. This is called *name hiding*.

There are two ways to address this. The first is to just use different variable names for the two. That's what we did above, though the names we chose are not great. A much more common convention in C# is to place an underscore before field names, as shown below:

```
class Score
{
    public string _name;
    public int _points;
    public int _level;

    public Score(string name, int points, int level)
    {
        _name = name;
        _points = points;
        _level = level;
    }
}
```

The underscores let us use similar names with a clear way to differentiate fields from local variables and parameters.

Using underscores is so common that it is the de facto standard for naming fields. You may also see some variations on the idea, such as using an **`m_`** or a **`my`** prefix. These conventions are used in other programming languages, and some programmers bring them into the C# world because they are familiar. But most C# programmers prefer the single underscore. What you choose is far less important than being *consistent*. You don't want fields named **`name`**, **`myPoints`**, **`level_`** and constructor parameters called **`_name`**, **`points`**, and **`my_level`**. You'll never keep them straight.

The second solution to name hiding is the **`this`** keyword. The **`this`** keyword is like a special variable that always refers to the object you are currently in. Using it, we can access fields directly, regardless of what names we have used for local variables and parameters:

```
class Score
{
    public string name;
    public int points;
    public int level;

    public Score(string name, int points, int level)
    {
        this.name = name;
        this.points = points;
        this.level = level;
    }
}
```

All three parameters hide fields of the same name, but we can still reach them using **`this`**. The **`this`** keyword allows us to use straightforward names without decoration while still allowing everything to work out. This approach is also popular among C# programmers. I'll follow the underscore convention in this book; it is the more common choice.

Calling Other Constructors with `this`

Sometimes, you'd like to reuse the code in one constructor from another. But you can't just call a constructor without using **`new`**, and if you did that in a constructor, you'd be creating a second object while creating the first, which isn't what you want. If you want one constructor to build off another one, use the **`this`** keyword:

```
class Score
{
    public string _name;
    public int _points;
    public int _level;

    public Score() : this("Unknown", 0, 1)
    {
    }

    public Score(string name, int points, int level)
    {
        _name = name;
        _points = points;
        _level = level;
    }
}
```

This allows one constructor to run another constructor first, eliminating duplicate code.

Leaving Off the Class Name

When you are creating new instances of a class, if the compiler has enough information to know which class you are using, you can leave the class name out:

```
Score first = new();
Score second = new("R2-D2", 12420, 15);
```

This is like **`var`**, only on the opposite side of the equals sign. The compiler can infer that you are creating an instance of the **`Score`** class because it is assigned to a **`Score`**-typed variable. This feature is most valuable when our type name is long and complex.

Object-Oriented Design

The concept of breaking large programs down into small parts, each managed by an object and all working together, is powerful. We will continue to learn about the mechanics and tools for doing this throughout Part 2.

The harder challenge is figuring out the right breakdown. Which objects should exist? Which classes should be defined? How do they work together? These questions are a topic called *object-oriented design*. Their answers are not always clear, even for veteran programmers. It is also a subject that deserves its own book (or dozens). But in a few levels (Level 23), we will get a crash course in object-oriented design to have a foundation to build on.

Challenge **Vin Fletcher's Arrows** **100 XP**

Vin Fletcher is a skilled arrow maker. He asks for your help building a new class to represent arrows and determine how much he should sell them for. "A tiny fragment of my soul goes into each arrow; I care not for the money; I just need to be able to recoup my costs and get food on the table," he says.

Each arrow has three parts: the arrowhead (steel, wood, or obsidian), the shaft (a length between 60 and 100 cm long), and the fletching (plastic, turkey feathers, or goose feathers).

His costs are as follows: For arrowheads, steel costs 10 gold, wood costs 3 gold, and obsidian costs 5 gold. For fletching, plastic costs 10 gold, turkey feathers cost 5 gold, and goose feathers cost 3 gold. For the shaft, the price depends on the length: 0.05 gold per centimeter.

Objectives:

- Define a new **`Arrow`** class with fields for arrowhead type, fletching type, and length. (**Hint:** arrowhead types and fletching types might be good enumerations.)
- Allow a user to pick the arrowhead, fletching type, and length and then create a new **`Arrow`** instance.
- Add a **`GetCost`** method that returns its cost as a **`float`** based on the numbers above, and use this to display the arrow's cost.

Level 19

Information Hiding

Speedrun

- Information hiding is where some details are hidden from the outside world while still presenting a public boundary that the outside world can still interact with.
- Class members should be marked **`public`** or **`private`** to indicate which of the two is intended.
- Data (fields) should be **`private`** in nearly all cases.
- Abstraction: when things are private, they can change without affecting the outside world. The outside world depends on the public parts, while anything private can change without problems.
- A third level is **`internal`**, which is meant to be used only inside the project.
- Classes and other types also have an accessibility level: **`public class X { ... }`**

This level covers the next two fundamental concepts of object-oriented programming: information hiding and abstraction.

We just saw that with encapsulation, an object could be responsible for a part of the system, containing its own data in special variables called fields, and provide its own list of abilities in the form of methods.

Our second principle is a simple extension of encapsulation (treated as the same by some):

Object-Oriented Principle #2: Information Hiding—Only the object itself should directly access its data.

To illustrate why this matters, consider the code below:

```
class Rectangle
{
    public float _width;
    public float _height;
    public float _area;

    public Rectangle(float width, float height, float area)
    {
        _width = width;
        _height = height;
        _area = area;
```

```
    }
}
```

This is a good beginning, but there is a problem brewing. A rectangle's area is defined as its width and height multiplied together. A rectangle with a length of 1 and a height of 1 has an area of 1. A rectangle with a length of 2 and a height of 3 has an area of 6. However, our current definition of **`Rectangle`** could allow this:

```
Rectangle rectangle = new Rectangle(2, 3, 200000);
```

Wouldn't it be nice if we could enforce this kind of rule? Removing the **`area`** parameter from the constructor and computing the area instead prevents somebody (including ourselves in 3 weeks when we forget the details) from accidentally supplying an illogical area.

```
public Rectangle(float width, float height)
{
    _width = width;
    _height = height;
    _area = width * height;
}
```

This ensures new rectangles always start with the correct area. But we still have a problem:

```
Rectangle rectangle = new Rectangle(2, 3);
rectangle._area = 200000;
Console.WriteLine(rectangle._area);
```

While the area is initially computed correctly, this code does not stop somebody from accidentally or intentionally changing the area. The outside world can reach in and mess with the rectangle's data in ways that shouldn't be allowed.

If the **`Rectangle`** class could keep its data hidden, the outside world could not put **`Rectangle`** instances into illogical or inconsistent states. Of course, the outside world will sometimes want to know about the rectangle's current size and area and may want to change its size. But all of that can be carefully protected through methods.

THE `PUBLIC` AND `PRIVATE` ACCESSIBILITY MODIFIERS

When we started making classes in the previous level, we slapped a **`public`** on all our fields and methods. This is the root of our information hiding problem because it makes it so the outside world can reach it.

Every member of a class—fields and methods alike—has an *accessibility level*. This level determines where the thing is accessible from. The **`public`** keyword gives the member public accessibility—usable anywhere. Instead of **`public`**, we could use **`private`**, which gives the member private accessibility—usable only within the class itself. The **`public`** and **`private`** keywords are both called *accessibility modifiers* because they change the accessibility level of the thing they are applied to. If we make our fields **`private`**, then the outside world cannot directly interfere with them:

```
class Rectangle
{
    private float _width;
    private float _height;
    private float _area;
```

```
    public Rectangle(float width, float height)
    {
        _width = width;
        _height = height;
        _area = width * height;
    }
}
```

Our data is now private. We can still use the fields inside the class as the constructor does to initialize them. But making them private ensures the outside world cannot change the area and create an inconsistent rectangle.

But now we have the opposite problem. We've sealed off all access to those fields. The outside world will want *some* visibility and perhaps some control over the rectangle. With all our fields marked **`private`**, we can no longer even do this:

```
Rectangle rectangle = new Rectangle(2, 3);
Console.WriteLine(rectangle._area); // DOESN'T COMPILE!
```

Since the outside world needs to know the rectangle's area, does that mean we must make the field public anyway? In general, no. Instead of allowing direct access to our fields, we provide controlled access through methods. For example, the outside world will want to know the rectangle's width, height, and area. So we add these methods to the **`Rectangle`** class:

```
public float GetWidth() => _width;
public float GetHeight() => _height;
public float GetArea() => _area;
```

The fields stay private, and the outside world can still get their questions answered without having unfettered access to the data.

If the outside world also needs to *change* the rectangle's dimensions, we can also solve that with methods:

```
public void SetWidth(float value)
{
    _width = value;
    _area = _width * _height;
}

public void SetHeight(float value)
{
    _height = value;
    _area = _width * _height;
}
```

We've decided it is reasonable to ask a rectangle to update its width and height and added methods for those. But we've decided we don't want to let people directly change the area, so we skip that one.

I intentionally chose names that start with **`Get`** and **`Set`**. Methods that retrieve a field's current value are called *getters*. Methods that assign a new value to a field are called *setters*. The above code shows that these methods allow us to perform more than just setting a new value for the field. Both **`SetWidth`** and **`SetHeight`** update the rectangle's area to ensure it stays consistent with its width and height.

These changes give us the following **`Rectangle`** class:

```
class Rectangle
{
    private float _width;
    private float _height;
    private float _area;

    public Rectangle(float width, float height)
    {
        _width = width;
        _height = height;
        _area = _width * _height;
    }

    public float GetWidth() => _width;
    public float GetHeight() => _height;
    public float GetArea() => _area;

    public void SetWidth(float value)
    {
        _width = value;
        _area = _width * _height;
    }

    public void SetHeight(float value)
    {
        _height = value;
        _area = _width * _height;
    }
}
```

With these changes, if we want to create a rectangle and change its size, we use the new methods instead of directly accessing its fields:

```
Rectangle rectangle = new Rectangle(2, 3);
rectangle.SetWidth(3);
Console.WriteLine(rectangle.GetArea());
```

Information hiding allows an object to protect its data. Each object is its own gatekeeper. If another object wants to see what state the object is in or change its state, it must request that information by calling a getter or setter method, rather than just reaching in and grabbing it. This way, objects can enforce rules about their data, as we see here with the rules around a rectangle's area.

As written above, information hiding came at the cost of substantially more complex code—the statement **`rectangle.SetWidth(3);`** is harder to understand than **`rectangle._width = 3;`**. Even if this were the end of the story, the benefits of information hiding would outweigh the added complexity costs. But it isn't the end of the story; we will see a better way to do this kind of stuff in Level 20. This solution is just a temporary one.

What if you don't have any rules to enforce? Is it okay to use public fields then? The principle of information hiding will nearly always prevent more pain than it causes. Even if you don't have any rules to enforce now, they usually arise as the program grows, and there are often more rules to enforce than might appear at first glance. For example, should our **`Rectangle`** class allow negative widths and heights? Arguably, that shouldn't be allowed, and our setter methods should check for it. But it is a guideline, and there are (rare!) exceptions.

The Default Accessibility Level is `private`

While we have intentionally put **`public`** or **`private`** on all our class members, this is not strictly necessary. We could leave it off entirely. If you don't specify an accessibility level, members of a class will be private.

In most cases, I recommend that you don't leave off the accessibility level; always put either **`public`** or **`private`** (or one of the other levels that we will learn later) on each class member. This forces you to think through how accessible the member ought to be. That exercise is worth the time it takes.

When to Use `private` and `public`

Two rules of thumb give us clues about whether to make things private or public.

The first, which we touched on earlier, is that a class should protect its data. Fields should almost always be **`private`**. There are exceptions, but these are rare.

The second is that things should always be as inaccessible as possible while allowing the class to fulfill its role in the system. For example, you could say that the getters and setters in our most recent **`Rectangle`** class definition are part of the job of representing a rectangle. So it is reasonable for each of those to be **`public`**. But three different times, we had a line of code that looked like **`_area = _width * _height;`**. We could make a method called **`UpdateArea()`** that contains this logic and then call it in three different spots (the constructor, **`SetWidth`**, and **`SetHeight`**). Should **`UpdateArea`** be private or public? Updating the area is not something the outside world should have to request specifically. It is details of how we have created the **`Rectangle`** class. Since the outside world doesn't need to do it, this new method would probably be better as a private method.

Sometimes, you'll get the accessibility level wrong. That's part of making software. Fortunately, you can change it later. But it is easier to take something private and make it public than the reverse. The outside world may already be using something initially made public, and you'd have to eliminate those.

Accessibility Levels as Guidelines, Not Laws

When you make something **`private`**, it does not mean the outside world has no possible way to use the code. It just means the compiler is enlisted to help ensure things intended to be kept hidden don't accidentally get used. It creates compiler errors when you attempt to misuse private members. However, there are ways to get around this; somebody creative and reckless enough can skirt the protections the compiler provides. (Reflection, described briefly in Level 47, is one such way.) It will ensure you don't accidentally shoot yourself in the foot but can't stop you from doing so intentionally.

ABSTRACTION

A magical thing happens when the principles of encapsulation and information hiding are followed. The inner workings of a class are not visible to the outside world. It is like a cell phone's insides: as long as the phone's buttons and screen work, we don't care how the circuitry on the inside works. The human body is like this, as well. We don't need to know how the nerves and tendons connect, as long as things are working correctly.

With the clear boundary provided by encapsulation and the inner workings kept secret through information hiding, those inner workings can change entirely without any visible effect on the outside world. This ability is called *abstraction* and is our third fundamental principle of object-oriented programming:

Object-Oriented Principle #3: Abstraction—The outside world does not need to know each object or class's inner workings and can deal with it as an abstract concept. Abstraction allows the inner workings to change without affecting the outside world.

That doesn't mean you can't poke around and see how things work on the inside. Curious minds will always do that. But if a class correctly does the job it advertises through its public members, you can put the details of *how* it works out of your mind. It also provides isolation from the rest of the world when working on the inside of a class. You can change anything you want that doesn't affect the public boundary, and the rest of the program won't be affected by it. You can swap out a battery in a cell phone or put artificial valves in the human heart, and the outside world won't be affected by it. Abstraction is essential in breaking down big problems into smaller ones because you can work on each part in isolation. You don't have to remember how every aspect of the entire program works to do anything. Once a class has been created, you can quit worrying about its details and use it as a cohesive whole.

Let's illustrate with an example. Earlier versions of our **`Rectangle`** class had a field for the rectangle's area, which got updated any time the width or height changed. But we can change this to compute the area as needed and ditch the field without affecting the rest of our program:

```
class Rectangle
{
    private float _width;
    private float _height;

    public Rectangle(float width, float height)
    {
        _width = width;
        _height = height;
    }

    public float GetWidth() => _width;
    public float GetHeight() => _height;
    public float GetArea() => _width * _height;

    public void SetWidth(float value) => _width = value;
    public void SetHeight(float value) => _height = value;
}
```

The **`_area`** field is gone, and **`SetWidth`**, **`SetHeight`**, and the constructor no longer calculate the area. Instead, it is calculated on demand when somebody asks for the area via **`GetArea`**. The outside world is oblivious to this change. They used to retrieve the rectangle's area through **`GetArea`** and still do.

Abstraction is a vital ingredient in building larger programs. It lets you make one piece of your program at a time without having to remember every detail as you go.

TYPE ACCESSIBILITY LEVELS AND THE `INTERNAL` MODIFIER

You can (and usually should) place accessibility levels on the types you define:

```
public class Rectangle
{
    // ...
}
```

For type definitions like this, **`private`** is not meaningful and is not allowed. It limits usage to just within the class, so it doesn't make sense to apply it to the whole class.

You might think that leaves **`public`** as the only option, but there is another: **`internal`**. Initially, you won't see many differences between **`public`** and **`internal`**. The difference is that things made **`public`** can be accessed everywhere, including in other projects, while **`internal`** can only be used in the project it is defined in. Consider, for example, all of the code in .NET's Base Class Library, like **`Console`** and **`Convert`**. That code is meant to be reused everywhere. **`Console`** and **`Convert`** are both **`public`**.

If you make a new type (class, enumeration, etc.) and feel that its role is a supporting role—details that help other classes accomplish their job, but not something you would want the outside world to know exists—you might choose to make this type **`internal`**.

Right now, we are building self-contained programs. We haven't made anything that we would expect other projects to reuse. You might be thinking, "I don't expect *any* of this to be reused by myself or anyone else. Why should I make anything **`public`**?"

Indeed, that is a legitimate thought process, and some would argue for making everything **`internal`** until you create something you specifically intend to reuse. It is a reasonable approach, and you can use it if you choose. But most C# programmers follow a somewhat different thought process.

There are three levels of share/don't-share decisions to make. (1) Do I share a project or not? (2) Should this individual type definition be shared or not? (3) Should this member—a field or a method—be shared or not? C# programmers usually consider these different levels in isolation. Suppose you are deciding whether to make something **`public`**, **`internal`**, or **`private`**. You assume that its container is as broadly available as possible and say, "If this thing's container were useable anywhere, how available should this specific item be?" For a class, you would say, "If this project were available to anybody, would I want them to be able to reuse this class? Or is this a secret detail that I'd want to reserve the right to change without affecting anybody?" For a method, "If this class were **`public`**, would I want this method to be **`public`**, or is this something I want to make less accessible so that I can change it more easily later?"

This second approach is more nuanced. It leads to more accessible things in less accessible things—a **`public`** class in a project you are not sharing, a **`public`** method in an **`internal`** class, etc. But it allows every accessibility decision to be made independent of every other accessibility decision. If you change a class from **`internal`** to **`public`** or vice versa, you don't need to reconsider which of its members should also change with it. The same is true if you decide to start or stop sharing the project as a whole.

This second approach leads to most types being **`public`**, many methods being **`public`**, and nearly all fields being **`private`**, with only a handful of **`internal`** types and methods, even for a project that is never reused.

I'm bringing up **`internal`** here because it is the default accessibility level for a type if none is explicitly written out. My advice is to always write out your intended accessibility level rather than leave it to the defaults. It forces you to build the habit of conscientiously deciding

accessibility levels instead of leaving it to coincidence. If you decide you prefer using the default in a few months, you can quit writing it out explicitly.

By the way, while type definitions must be either **`public`** or **`internal`**, members of a class can be **`public`**, **`private`**, or **`internal`**. (For an enumeration, members are automatically public, and you cannot change that.)

The compiler ensures that you cohesively use accessibility levels and flags inconsistencies as compiler errors. For example, if you have a **`public`** class with a **`public`** method whose return type is an **`internal`** class, the compiler will report it as an error. This method would inadvertently publicly leak something you indicated should be **`internal`**.

Challenge **Vin's Trouble** **50 XP**

"Master Programmer!" Vin Fletcher shouts at you as he races to catch up to you. "I have a problem. I created an arrow for a young man who took it and changed its length to be half as long as I had designed. It no longer fit in his bow correctly and misfired. It sliced his hand pretty bad. He'll survive, but is there any way we can make sure somebody doesn't change an arrow's length when they walk away from my shop? I don't want to be the cause of such self-inflicted pain." With your knowledge of information hiding, you know you can help.

Objectives:

- Modify your **`Arrow`** class to have **`private`** instead of **`public`** fields.
- Add in getter methods for each of the fields that you have.

Level 20

Properties

Speedrun

- Properties give you field-like access while still protecting data with methods: `public float Width { get => width; set => width = value; }`. To use a property: `rectangle.Width = 3;`
- Auto-properties are for when no extra logic is needed: `public float Width { get; set; }`
- Properties can be read-only, only settable in a constructor: `public float Width { get; }`
- Fields can also be read-only: `private readonly float _width = 3;`
- With properties, objects can be initialized using object initializer syntax: `new Rectangle() { Width = 2, Height = 3 }`.
- An `init` accessor is like a setter but only usable in object initializer syntax. `public float Width { get; init; }`

The Basics of Properties

While information hiding has significant benefits, it adds complexity to our code. Instead of a simple class with three public fields and a constructor, we ended up with this:

```
public class Rectangle
{
    private float _width;
    private float _height;

    public Rectangle(float width, float height)
    {
        _width = width;
        _height = height;
    }

    public float GetWidth() => _width;
    public float GetHeight() => _height;
    public float GetArea() => _width * _height;

    public void SetWidth(float value) => _width = value;
```

```
    public void SetHeight(float value) => _height = value;
}
```

And instead of **`rectangle._width = 3;`** we ended up with **`rectangle.SetWidth(3);`**.

But we had rules we needed to enforce and wanted to preserve the benefits of abstraction to change the inner workings without affecting anything else. Those two things pushed us to this more complex version of the code.

But in C#, there is a tool we can use to get the benefits of both information hiding and abstraction while keeping our code simple: properties. A *property* pairs a getter and setter under a shared name with field-like access.

Consider the three elements that dealt with the rectangle's width above:

```
private float _width;

public float GetWidth() => _width;

public void SetWidth(float value) => _width = value;
```

To swap this out for a property, we would write the following code:

```
private float _width;

public float Width
{
    get => _width;
    set => _width = value;
}
```

This defines a property with the name **`Width`** whose type is **`float`**. Properties are another type of member that we can put in a class. They have their own accessibility level. I made this one **`public`** since the equivalent methods, **`GetWidth`** and **`SetWidth`** were **`public`**. Each property has a type. This one uses **`float`**. After modifiers and the type is the name (**`Width`**). Note the capitalization. It is typical to use UpperCamelCase for property names.

The body of a property is defined with a set of curly braces. Inside that, you can define a getter (with the **`get`** keyword) and a setter (with the **`set`** keyword), each with its own body. The above code used expression bodies, but you can also use block bodies for either or both:

```
public float Width
{
    get
    {
        return _width;
    }
    set
    {
        _width = value;
    }
}
```

In this case, the expression body is simpler. In other situations, you'll need a block body.

The getter is required to return a value of the same type as the property (**`float`**). The setter has access to the special **`value`** variable in its body. We didn't define a **`value`** parameter, but in essence, one automatically exists in a property setter.

Many properties provide logic around accessing a single field, as the **`Width`** does with **`_width`**. In these cases, the field is called the property's *backing field* or *backing store*. In most situations, the property and its backing field share the same name, aside from underscores and capitalization, which helps you track which property is tied to which field.

Properties do not require both getters and setters. You can have a **`get`**-only property or a **`set`**-only property. A **`get`**-only property makes sense for something that can't be changed from the outside. The rectangle's area is like this. We could make a **`get`**-only property for it:

```
public float Area
{
    get => _width * _height;
}
```

If a property is **`get`**-only and the getter has an expression body, we can simplify it further:

```
public float Area => _width * _height;
```

Thus, the first stab at a property-based **`Rectangle`** class might look like this:

```
public class Rectangle
{
    private float _width;
    private float _height;

    public Rectangle(float width, float height)
    {
        _width = width;
        _height = height;
    }

    public float Width
    {
        get => _width;
        set => _width = value;
    }

    public float Height
    {
        get => _height;
        set => _height = value;
    }

    public float Area => _width * _height;
}
```

The most significant benefit comes in the outside world, which now has field-like access to the properties instead of method-like access:

```
Rectangle r = new Rectangle(2, 3);
r.Width = 5;
Console.WriteLine($"A {r.Width}x{r.Height} rectangle has an area of {r.Area}.");
```

In the code above, the line **`r.Width = 5;`** will call the **`Width`** property's setter, and the special **`value`** variable will be **`5`** when the setter code runs.

On the final line, referencing the **`Width`**, **`Height`**, and **`Area`** properties will call the getters for each of those properties.

Our code can use clean, simple syntax without giving up information hiding and abstraction!

A property's getter and setter do not need to have the same accessibility level. Either getter or setter can reduce accessibility from what the property has. If we want the property to have a public getter and a private setter, we could do this:

```
public float Width
{
    get => _width;
    private set => _width = value;
}
```

Auto-Implemented Properties

Some properties will have complex logic for its getter, setter, or both. But others do not need anything fancy and end up looking like this:

```
public class Player
{
    private string _name;

    public string Name
    {
        get => _name;
        set => _name = value;
    }
}
```

Because these are commonplace, there is a concise way to define properties of this nature called an *auto-implemented property* or an *auto property*:

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

You don't define bodies for either getter or setter, and you don't even define the backing field. You just end the getter and setter with a semicolon. The compiler will generate a backing field for this property and create a basic getter and setter method around it.

The backing field is no longer directly accessible in your code, but that's rarely an issue. However, one problematic place is initializing the backing field to a specific starting value. We can still solve that with an auto-property like this:

```
public string Name { get; set; } = "Player";
```

Don't forget the semicolon at the end of the line! It won't compile if you forget it.

A version of the **`Rectangle`** class that uses auto-properties might look like this:

```
public class Rectangle // Note how short this code got with auto-properties.
{
    public float Width { get; set; }
    public float Height { get; set; }
    public float Area =>  Width * Height;

    public Rectangle(float width, float height)
    {
        Width = width;
```

```
        Height = height;
    }
}
```

IMMUTABLE FIELDS AND PROPERTIES

Auto-properties can be get-only, like a regular property. (They cannot be set-only; there is no scenario where that is useful as it would be a black hole for data.) This makes the property *immutable*, "im-" meaning "not" and "mutable," meaning changeable. When a property is get-only, it can still be assigned values, but only from within a constructor. These are also sometimes referred to as *read-only properties*. When a property is immutable, its behavior is like concrete or a tattoo. You have complete control when the object is being created, but it cannot be changed again once the object is created.

Consider this version of the **`Player`** class, which has made **`Name`** immutable:

```
public class Player
{
    public string Name { get; } = "Player 1";

    public Player(string name)
    {
        Name = name;
    }
}
```

The getter is public, so we can always retrieve **`Name`**'s current value. And even without a setter, we can still assign a value to **`Name`** in an initializer or constructor. But after creation, we cannot change **`Name`** from inside or outside the class.

While this sounds restrictive, there are many benefits to immutability. For example, we spent a lot of time worrying about our **`Rectangle`** class's area becoming inconsistent with its width and height. If we made all of **`Rectangle`**'s properties immutable and only gave them values in the constructor, there would be no possible way for the data to become inconsistent afterward.

If immutable properties are beneficial, what about fields? If you have a field that you don't want to change after construction, you can apply the **`readonly`** keyword to it as a modifier:

```
public class Player
{
    private readonly string _name;

    public Player(string name)
    {
        _name = name;
    }
}
```

Like immutable properties, this can be assigned a value inline as an initializer or in a constructor, but nowhere else.

When all of a class's properties and fields are immutable (**`get`**-only auto-properties and **`readonly`** fields), the entire object is immutable. Not every object should be made immutable. But when they can be, they are much easier to work with because you know the object cannot change.

Object Initializer Syntax and Init Properties

While constructors should get the object into a good starting state, some initialization is best done immediately *after* the object is constructed, changing the values of a handful of properties right after construction. It is like making some final adjustments as the concrete is still drying. Let's say we have this **`Circle`** class:

```
public class Circle
{
    public float X { get; set; } = 0; // The x-coordinate of the circle's center.
    public float Y { get; set; } = 0; // The y-coordinate of the circle's center.
    public float Radius { get; set; } = 0;
}
```

With this definition, we could make a new circle and set its properties like this:

```
Circle circle = new Circle();
circle.Radius = 3;
circle.X = -4;
```

C# provides a simple syntax for setting properties right as the object is created called *object initializer syntax*, shown below:

```
Circle circle = new Circle() { Radius = 3, X = -4 };
```

If the constructor is parameterless, you can even leave out the parentheses:

```
Circle circle = new Circle { Radius = 3, X = -4 };
```

You cannot use object initializer syntax with properties that are **`get`**-only. While you can assign a value to them in the constructor, object initializer syntax comes after the constructor finishes. This is a predicament because it would mean you must make your properties mutable (have a setter) to use them in object initializer syntax, which is too much power in some situations.

The middle ground is an **`init`** accessor. This is a setter that can be used in limited circumstances, including with an inline initializer (the **`0`**'s below) and in the constructor, but also in object initializer syntax:

```
public class Circle
{
    public float X { get; init; } = 0;
    public float Y { get; init; } = 0;
    public float Radius { get; init; } = 0;
}
```

Which can be used like this:

```
Circle circle = new Circle { X = 1, Y = 4, Radius = 3 };

// This would not compile if it were not a comment:
// circle.X = 2;
```

Challenge **The Properties of Arrows** **100 XP**

Vin Fletcher once again has run to catch up to you for help with his arrows. "My apologies, Programmer! This will be the last time I bother you. My cousin, Flynn Vetcher, is the only other arrow maker in the area. He doesn't care for his craft and makes wildly dangerous and overpriced arrows. But people keep buying them because they think my **`GetLength()`** methods are harder to work with than his public

`_length` fields. I don't want to give up the protections we just gave these arrows, but I remembered you saying something about properties. Maybe you could use those to make my arrows easier to work with?"

Objectives:

- Modify your **`Arrow`** class to use properties instead of **`GetX`** and **`SetX`** methods.
- Ensure the whole program can still run, and Vin can keep creating arrows with it.

ANONYMOUS TYPES

Using object initializer syntax and **`var`**, you can create new types that don't even have a formal name or definition—an *anonymous type*.

```
var anonymous = new { Name = "Steve", Age = 34 };
Console.WriteLine($"{anonymous.Name} is {anonymous.Age} years old.");
```

This code creates a new instance of an unnamed class with two **`get`**-only properties: **`Name`** and **`Age`**. Since this type doesn't have a name, you must use **`var`**. You can only use anonymous types within a single method. You cannot use one as a parameter, return type, or field.

Anonymous types have the occasional use but don't underestimate the value of just creating a small, simple class for what you are doing (giving things a name is valuable) or using a tuple.

Level 21

Static

Speedrun

- Static things are owned by the type rather than a single instance (shared across all instances).
- Fields, methods, and constructors can all be static.
- If a class is marked static, it can only contain static members (`Console`, `Convert`, `Math`).

Static Members

By this point, you may have noticed an inconsistency. We have used `Console`, `Convert`, and `Math` but have never done `new Console()`. We have used our own classes differently.

In C#, class members naturally belong to instances of the class. Consider this simple example:

```
public class SomeClass
{
    private int _number;
    public int Number => _number;
}
```

Each instance of `SomeClass` has its own `_number` field, and calling methods or properties like the `Number` property is associated with specific instances and their individual data. Each instance is independent of the others, other than sharing the same class definition.

But you can also mark members of a class with the `static` keyword to detach them from individual instances and tie it to the class itself. In Visual Basic, the equivalent keyword is `Shared`, which is a more intuitive name.

All member types that we have seen so far can be made static.

Static Fields

By applying the `static` keyword to a field, you create a *static field* or *static variable*. These are especially useful for defining variables that affect every instance in the class. For example,

we can add these two static fields that will help determine if a score is worth putting on the high score table:

```
public class Score
{
    private static readonly int PointThreshold = 1000;
    private static readonly int LevelThreshold = 4;

    // ...
}
```

Earlier, we saw that C# programmers usually name fields with _lowerCamelCase, but if they are static, they tend to be UpperCamelCase instead.

These two fields are **`private`** and **`readonly`**, but we can use all the same modifiers on a static field as a normal field. Occasionally, regular, non-static fields are referred to as *instance fields* when you want to make a clear distinction.

Static fields are used within the class in the same way that you would use any other field:

```
public bool IsWorthyOfTheHighScoreTable()
{
    if (Points < PointThreshold) return false;
    if (Level < LevelThreshold) return false;
    return true;
}
```

If a static field is public, it can be used outside the class through the class name (**`Score.PointThreshold`**, for example).

Global State

Static fields have their uses, but a word of caution is in order. If a field is static, public, and not read-only, it creates *global state*. Global state is data that can be changed and used anywhere in your program. Global state is considered dangerous because one part of your program can affect other parts even though they seem unrelated to each other. Unexpected changes to global state can lead to bugs that take a long time to figure out, and in most situations, you're better off not having it.

It is the combination that is dangerous. Making the field **`private`** instead of **`public`** limits access to just the class, which is easier to manage. Making the field **`readonly`** ensures it can't change over time, preventing one part of the code from interfering with other parts. If it is not static, only parts of the program that have a reference to the object will be able to do anything with it. Just be cautious any time you make a **`public static`** field.

Static Properties

Properties can also be made static. These can use static fields as their backing fields, or you can make them auto-properties. These have the same global state issue that fields have, so be careful with **`public static`** properties as well.

Below is the property version of those two thresholds that we made as fields earlier:

```
public class Score
{
    public static int PointThreshold { get; } = 1000;
    public static int LevelThreshold { get; } = 4;
```

```
    // ...
}
```

We use static properties on the **`Console`** class. **`Console.ForegroundColor`** and **`Console.Title`** are examples. **`Console.ForegroundColor`** is a good example of the danger of global state. If one part of the code changes the color to red to display an error, everything afterward will also be written in red until somebody changes it back.

Static Methods

Methods can also be static. A static method is not tied to a single instance, so it cannot refer to any non-static (instance) fields, properties, or methods.

Static methods are most often used for utility or helper methods that provide some sort of service related to the class they are placed in, but that isn't tied directly to a single instance. For example, the following method determines how many scores in an array belong to a specific player:

```
public static int CountForPlayer(string playerName, Score[] scores)
{
    int count = 0;
    foreach (Score score in scores)
        if (score.Name == playerName) count++;
    return count;
}
```

This method would not make sense as an instance method because it is about many scores, not a single one. But it makes sense as a static method in the **`Score`** class because it is closely tied to the **`Score`** concept.

Another common use of static methods is a *factory method*, which creates new instances for the outside world as an alternative to calling a constructor. For example, this method could be a factory method in our **`Rectangle`** class:

```
public static Rectangle CreateSquare(float size) => new Rectangle(size, size);
```

This method can be called like this:

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

This code also illustrates how to invoke static members from outside the class. But it should look familiar; this is how we've been calling things like **`Console.WriteLine`** and **`Convert.ToInt32`**, which are also static methods.

Static Constructors

If a class has static fields or properties, you may need to run some logic to initialize them. To address this, you could define a static constructor:

```
public class Score
{
    public static readonly int PointThreshold;
    public static readonly int LevelThreshold;

    static Score()
    {
        PointThreshold = 1000;
        LevelThreshold = 4;
```

```
    }

    // ...
}
```

A static constructor cannot have parameters, nor can you call it directly. Instead, it runs automatically the first time you use the class. Because of this, you cannot place an accessibility modifier like **`public`** or **`private`** on it.

STATIC CLASSES

Some classes are nothing more than a collection of related utility methods, fields, or properties. **`Console`**, **`Convert`**, and **`Math`** are all examples of this. In these cases, you may want to forbid creating instances of the class, which is done by marking it with the **`static`** keyword:

```
public static class Utilities
{
    public static int Helper1() => 4;
    public static double HelperProperty => 4.0;
    public static int AddNumbers(int a, int b) => a + b;
}
```

The compiler will ensure that you don't accidentally add non-static members to a static class and prevent new instances from being created with the **`new`** keyword. Because **`Console`**, **`Convert`**, and **`Math`** are all static classes, we never needed—nor were we allowed—to make an instance with the **`new`** keyword.

Challenge **Arrow Factories** **100 XP**

Vin Fletcher sometimes makes custom-ordered arrows, but these are rare. Most of the time, he sells one of the following standard arrows:

- The Elite Arrow, made from a steel arrowhead, plastic fletching, and a 95 cm shaft.
- The Beginner Arrow, made from a wood arrowhead, goose feathers, and a 75 cm shaft.
- The Marksman Arrow, made from a steel arrowhead, goose feathers, and a 65 cm shaft.

You can make static methods to make these specific variations of arrows easy.

Objectives:

- Modify your **`Arrow`** class one final time to include static methods of the form **`public static Arrow CreateEliteArrow() { ... }`** for each of the three above arrow types.
- Modify the program to allow users to choose one of these pre-defined types or a custom arrow. If they select one of the predefined styles, produce an **`Arrow`** instance using one of the new static methods. If they choose to make a custom arrow, use your earlier code to get their custom data about the desired arrow.

Level 22

Null References

Speedrun

- Reference types may contain a reference to nothing: **`null`**, representing a lack of an object.
- Carefully consider whether null makes sense as an option for a variable and program accordingly.
- Check for null with **`x == null`**, the null conditional operators **`x?.DoStuff()`** and **`x?[3]`**, and use **`??`** to allow null values to fall back to some other default: **`x ?? "empty"`**

Reference type variables like **`string`**, arrays, and classes don't store their data directly in the variable. The variable holds a reference and the data lives on the heap somewhere. Most of the time, these references point to a specific object, but in some cases, the reference is a special one indicating the absence of a value. This special reference is called a *null reference*. In code, you can indicate a null reference with the **`null`** keyword:

```
string name = null;
```

Null references are helpful when it is possible for there to be no data available for something. Imagine making a game where you control a character that can climb into a vehicle and drive it around. The vehicle may have a **`Character _driver`** field that can point out which character is currently in the driver's seat. The driver's seat might be empty, which could be represented by having **`_driver`** contain a null reference. **`null`** is the default value for reference types.

But null values are not without consequences. Consider this code:

```
string name = null;
Console.WriteLine(name.Length);
```

This code will crash because it tries to get the **`Length`** on a non-existent string. Spotting this flaw is easy because **`name`** is always **`null`**; it is less evident in other situations:

```
string name = Console.ReadLine(); // Can return null!
Console.WriteLine(name.Length);
```

Did **`ReadLine`** give us an actual string instance or **`null`**? You have probably not have encountered it yet, but there are certain situations where **`ReadLine`** can return null. (Try

pressing **Ctrl + Z** when the computer is waiting for you to enter something.) The mere possibility that it *could* be null requires us to proceed with caution.

NULL OR NOT?

For reference-typed variables, stop and think if null should be an option. If null is allowed, you will want to check it for null before using its members (methods, properties, fields, etc.). If null is not allowed, you will want to check any value you assign to it to ensure you don't accidentally assign null to it. We'll see several ways to check for null in a moment.

After deciding if a variable should allow null, we want to indicate this decision in our code. Any reference-typed variable can either have a **?** at the end or not. A **?** means that it may legitimately contain a null value. For example:

```
string? name = Console.ReadLine(); // Can return null!
```

In the code above, **`name`**'s type is now **`string?`**, which indicates it can contain any legitimate **`string`** instance, but it may also be null. Without the **?**, as we've done until now, we show that null is not an option.

Until now, we've been ignoring the possibility of null. There's even a good chance you've come away unscathed. In all the code we've seen so far, the only real threat has been that **`Console.ReadLine()`** might return null, and we haven't been accounting for it. However, you probably haven't been pressing **Ctrl + Z**, so it probably hasn't come up. Even if you did, we've usually taken our input and either displayed it directly or converted it to another type, and both **`Console.WriteLine`** and **`Convert.ToInt32`** (and its other methods) safely handle null.

But from now on, we're far more likely to encounter problems related to null, so it's time to start being more careful and making an intentional choice for each reference-typed variable about whether null should be allowed or not.

If we correctly apply (or skip) the **?** to our variables, we'll be able to get the compiler's help to check for null correctly. This help is immensely valuable. It is easy to miss something on your own. With the compiler helping you spot null-related issues, you won't miss much. Of course, the second benefit is that the code clearly shows whether null is a valid option for a variable. That is helpful to programmers (including yourself) who later look at your code.

Our examples have only used strings so far, but this applies to all reference types, including arrays and any class you make. We could (and should!) do a similar thing for usages of our **`Score`** and **`Rectangle`** classes.

Disabling Nullable Type Warnings

Annotating a variable with **?** is a relatively new feature of C# (starting in C# 9). If you look at older C# code (including most Internet code), you won't see any **?** symbols on reference-typed variables. All reference-typed variables were assumed to allow null as an option, and the compiler didn't help you find places where null might cause problems.

I don't recommend it, but if you want (or have a need) to go back to the old way, you can turn this feature off. This article describes how: **csharpplayersguide.com/articles/disable-null-checking**.

CHECKING FOR NULL

Once you take null references into account, you'll find yourself needing to check for null often. The mechanics of checking for null is quite simple. The easiest way is to compare a reference against the **`null`** literal, which is called a *null check*:

```
string? name = Console.ReadLine();
if (name != null)
    Console.WriteLine("The name is not null.");
```

If a variable indicates that null is an option, you will want to do a null check before using its members. If a variable indicates that null is not an option, you will want to do a null check on any value you're about to assign to the variable to ensure you don't accidentally put a null in it.

It is important to point out that, once compiled, there isn't a difference between **`string?`** and **`string`**. If you ignore the compiler warnings that are trying to help you get it right, even a plain **`string`** (without the **`?`**) can still technically hold a null value. Look for these compiler warnings and fix them by adding appropriate null checking or correctly marking a variable as allowing or forbidding null.

Null-Conditional Operators: `?.` and `?[]`

One problem with null checking is that there may be implications down the line. For example:

```
private string? GetTopPlayerName()
{
    return _scoreManager.GetScores()[0].Name;
}
```

`_scoreManager` could be null, **`GetScores()`** could return null, or the array could contain a null reference at index 0. If any of those are null, it will crash. We need to check at each step:

```
private string? GetTopPlayerName()
{
    if (_scoreManager == null) return null;

    Score[]? scores = _scoreManager.GetScores();
    if (scores == null) return null;

    Score? topScore = scores[0];
    if (topScore == null) return null;

    return topScore.Name;
}
```

The null checks make the code hard to read. They obscure the interesting parts.

There is another way: *null-conditional operators*. The **`?.`** and **`?[]`** operators can be used in place of **`.`** and **`[]`** to simultaneously check for null and access the member:

```
private string? GetTopPlayerName()
{
    return _scoreManager?.GetScores()?[0]?.Name;
}
```

Both **`?.`** and **`?[]`** evaluate the part before it to see if it is null. If it is, then no further evaluation happens, and the whole expression evaluates to **`null`**. If it is not null, evaluation will continue

as though it had been a normal `.` or `[]` operator. So if `_scoreManager` is null, then the above code returns a null value without calling `GetScores`. If `GetScores()` returns null, the above code returns a null without accessing index 0.

These operators do not cover every null-related scenario—you will sometimes need a good old-fashioned `if (x == null)`—but they can be a simple solution in many scenarios.

The Null Coalescing Operator: ??

The *null coalescing operator* (`??`) is also a useful tool. It takes an expression that might be null and provide a value or expression to use as a fallback if it is:

```
private string GetTopPlayerName() // No longer needs to allow nulls.
{
    return _scoreManager?.GetScores()?[0]?.Name ?? "(not found)";
}
```

If the code before the `??` evaluates to null, then the fallback value of `"(not found)"` will be used instead.

There is also a compound assignment operator for this:

```
private string GetTopPlayerName()
{
    string? name = _scoreManager?.GetScores()?[0]?.Name;
    name ??= "(not found)";
    return name;      // No compiler warning. `??=` ensures we have a real value.
}
```

The Null-Forgiving Operator: !

The compiler is pretty thorough in analyzing what can and can't be null and giving you appropriate warnings. On infrequent occasions, you know something about the code that the compiler simply can't infer from its analysis. For example:

```
string message = MightReturnNullIfNegative(+10);
```

Assuming the return type of `MightReturnNullIfNegative` is `string?`, the compiler will flag this as a situation where you are assigning a potentially null value to a variable that indicates null isn't allowed. But assuming the method name isn't a lie (which isn't always a safe assumption), we know the returned value can't be null.

To get rid of the compiler warning, we can use the *null-forgiving operator*: `!`. (C# uses this same symbol for the Boolean *not* operator, as we saw earlier in the book.) This operator tells the compiler, "I know this looks like a potential null problem, but it won't be. Trust me."

Using it looks like this:

```
string message = MightReturnNullIfNegative(+10)!;
```

You place it at the end of an expression that might be null to tell the compiler that it won't actually evaluate to null. With the `!` in there, the compiler warning will go away.

There's a danger to this operator. You want to be sure you're right. I've had times where I thought the compiler was wrong, and I knew better, but after studying the code a bit more, I realized the compiler was catching things I had missed. Use `!` sparingly, but use it when needed.

Level 23

Object-Oriented Design

Speedrun

- Object-oriented design is figuring out which objects should exist in your program, which classes they belong to, what responsibilities each should have, and how they should work together.
- The design starts with identifying the requirements of what you are building.
- Noun extraction helps get the design process started by identifying concepts and jobs to do in the requirements.
- CRC cards are a tool to think through a design with physical cards for each object, containing their class, responsibilities, and collaborators.
- Object-oriented design is hard, but you don't have to figure out the entire program all at once, nor do you have to get it right the first time.

As we tackle larger problems, our solutions grow in size as well. Objects allow us to take the entire problem and break it into small pieces, where each piece—each object—has its job in the overall system. Many objects—each doing their part and coordinating with the other objects—allow us to solve the overall problem in small pieces.

Object-oriented design is the part of crafting software where we decide:

- which objects should exist in our program,
- the classes each of those objects belong to,
- what responsibilities each class or object should handle,
- when objects should come into existence,
- when objects should go out of existence,
- which objects must collaborate with or rely upon which other objects,
- and how an object knows about the other objects it works with.

Object-oriented design is a vast topic that deserves its own book (or ten) and can take years to truly master. The focus of this book is the C# programming language, not object-oriented design. Yet if you don't know the basics of programming with objects (object-oriented

programming) and know how to structure your program to use them (object-oriented design), you will have difficulty making large programs. You won't get all the benefits that come from objects and classes in C#. While this level is not a complete guide, it is a starting point in that journey.

Object-oriented design is sometimes referred to by the simpler terms *software design* or *design*; you will see those terms used in this level and book to mean the same thing.

If there is one thing you should know about object-oriented design, it is that you are going to get it wrong sometimes. Even after 15 years of professional programming, I still look around after a few days or weeks of programming and realize I took the wrong path. The good news is that software is soft; it can always be changed. Unlike pouring concrete for a bridge, it is never too late to switch a design in software. This softness provides a sense of safety and freedom. You can never be irrevocably wrong with software. You just need to be willing to recognize that there might be a better path and be ready to change it.

You should also know that programs are not designed in a design Big Bang before typing out a single line of code (aside from programs like Hello World). More experience may let you work on larger chunks, but software is built a slice at a time and evolves as you create it. So don't fret over having to solve gigantic problems all at once; not even the pros do that.

As we go through this, we will use the classic game of *Asteroids* as an example. If you are not familiar with this game, look it up online and play it for a bit. Playing the game will help the examples in this level make more sense. We will be focusing on design elements, not drawing this game on the screen. (You could technically draw this in the console window, but that is far from ideal.)

Requirements

The first step of building object-oriented systems is understanding what the software needs to do. This is sometimes called *requirements gathering*, though that word has baggage. To many people, "gathering requirements" means spending weeks rehashing the same dry, dusty Word documents replete with proclamations like "THE SOFTWARE SHALL THIS" and "THE SOFTWARE SHALL THAT," with far too much detail here, far too little detail there, and conflicting details throughout. You may find yourself doing requirements this way someday, but something much simpler is usually sufficient.

Things like homework assignments and challenges in this book typically come with detailed requirements in their descriptions. In other cases, you may have to hunt down or invent the requirements yourself. I recommend putting these requirements—what the software needs to do—into words. Whether that is on paper, whiteboard, or digital document, the act of writing it out forces you to describe what you mean. Without this, the human brain likes to play this trick on you where it says, "I know this," and skips past the part where it proves that it knows it. (Besides, if you are working with others, you will need to do this so that everybody is on the same page.)

The simplest solution is to write out a sentence or two describing each feature. For example, a couple of requirements for the game of *Asteroids* could be "Asteroids drift through space at some specific velocity," and "When a bullet hits an asteroid, the asteroid is removed from the game."

For some things, a picture or illustration is a better way to show intent, so don't be afraid to sketch something out to support your short sentences. Quality doesn't matter in this situation; you do not need to be an artist.

You can also augment these short sentences with specific, concrete examples. Examples help you discover details that might have otherwise been missed and help ensure everybody understands things the same way. "An asteroid is at the coordinates (2, 4) with a velocity of (-1, 0). After 1 second passes, its coordinates should be (1, 4)." Even this single example shows that positions and velocities are measured in two directions (side-to-side and top-to-bottom) and that velocities are measured in units per second.

You do not need to collect every single requirement before moving forward. Software is best built a little at a time because your plans for the software evolve as they come together. You can sometimes benefit by having a long-term view of what might be needed later, but those long-term plans nearly always change. (There are situations where change is rare and knowing more details ahead of time is more beneficial. But these are rare.)

Designing the Software

Once we have identified the next thing to build through writing, supported with pictures and examples, we are ready to begin design. There are many ways to approach design. We will touch on a few, though programmers use a wide variety of techniques. Find a system that works well for you.

Noun Extraction

A possible first step is to identify the concepts and jobs that the requirements reveal. Concepts that appear in the requirements will often lead to classes of objects in your design. Jobs or tasks that appear in the requirements will often lead to responsibilities that your software must be able to do. Some object in your design must eventually handle that responsibility.

You can start this process by highlighting the nouns (and noun phrases) and verbs (and verb phrases) that appear in the requirements. This is called *noun extraction* or *noun and verb extraction*. It can be a good first step, but it is not magic. Not all nouns deserve to be classes in our program and not every important concept is explicitly stated in our requirements. It usually involves more work to discover which concepts and tasks are involved. But if you miss something, you can always change it later.

Let's look closely at this requirement: *Asteroids drift through space at some specific velocity*. The nouns *asteroid*, *space*, and *velocity* are all potential concepts that we may or may not make classes around, and the verb *drift* is a job that some object (or several objects) in our system will need to do. We may have some thoughts on how we could start designing our program from this.

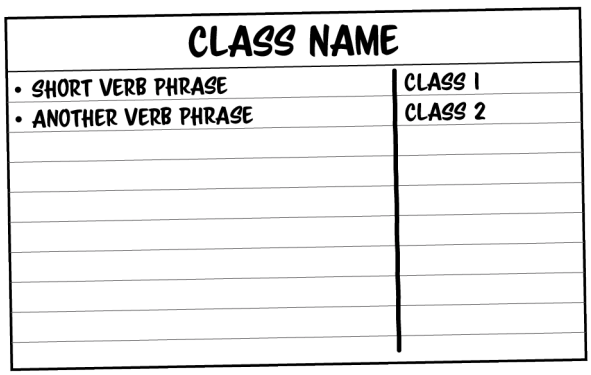
While we may use noun extraction (or the other tools described here) to come up with the beginnings of a potential design—a guess about the design—you are not done designing until you have code that solves the problem and whose structure is something you can live with. In that sense, the code itself is the only accurate representation of your design. But most programmers will begin exploring design options in lighter weight and more flexible tools than actual code, such as a whiteboard or pen and paper. With a whiteboard or pen and paper, change is trivial.

UML

Before moving on to the tool we will spend most of our time on, I must mention another. There is a very formal diagramming tool called the *Unified Modeling Language*, or *UML*. Many programmers around the world use this, and it is helpful to know it. However, it is a complicated system that is not ideal for new programmers. It is complex enough that even many experienced programmers prefer simpler tools when discussing design possibilities. I mention this so that you are aware of a tool that most developers know of and that many use. The technique we will see below (CRC cards) is far less formal and much lighter. I find it a helpful tool for people beginning with software design while still being meaningful for experienced object-oriented designers. But my experience has been that more programmers know about UML than CRC cards.

CRC Cards

CRC cards are a way to think through potential object-oriented designs and flesh out some detail. It helps you figure out which objects should exist, what parts of the overall problem each object should solve, and how they should work together. The short description of CRC cards is that you get a stack of blank 3x5 cards (or something similar) and create one card per object in your system. On each card, you will list three things: (1) the *class* that the object belongs to, written at the top, (2) the *responsibilities* that the object has in a list on the left side, and (3) the object's *collaborators*—other objects that help the object fulfill its responsibilities. CRC is short for Class-Responsibility-Collaborator. A sample CRC card might look like this:



Class names should be nouns or noun phrases. A good name gives you and others a simple way to refer to each type of object and is worth spending some time identifying a good name.

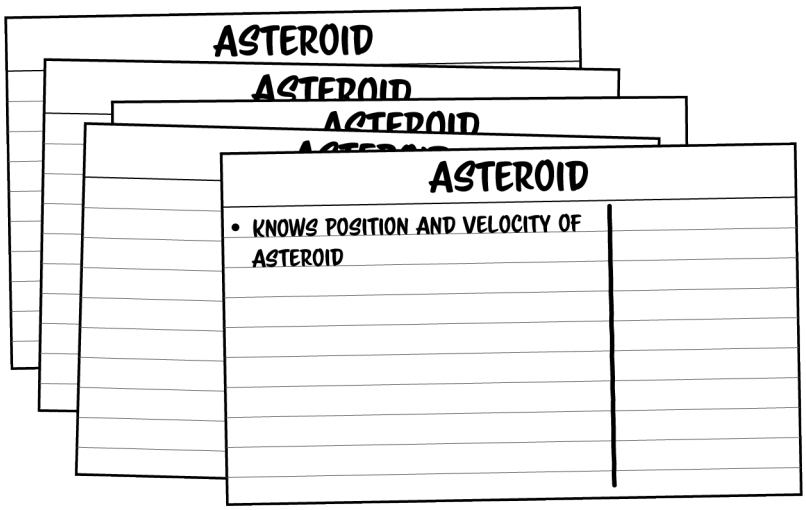
Each responsibility should be listed as a verb or verb phrase. If you run out of space on a card, you are probably asking it to do too much. Split its responsibilities into other cards and objects. A responsibility can be a thing to know or a thing to do. However, you should describe what the job is, not how to do it. Remember that each object needs the capacity to fulfill its responsibilities. It will need to know the data for its job, be handed the data in a method call, or ask its collaborators for it.

The collaborators of an object are the names of other classes that this object needs to fulfill its responsibilities. You could also use the word "helpers" here if you like that better. Just because one object uses another as a collaborator does not require that the relationship go both ways. One object can rely on another without the second object even knowing about the first.

Making CRC cards usually starts with the parts you know the best—the most obvious objects you will need. You then walk through different "what if" scenarios and talk through how your

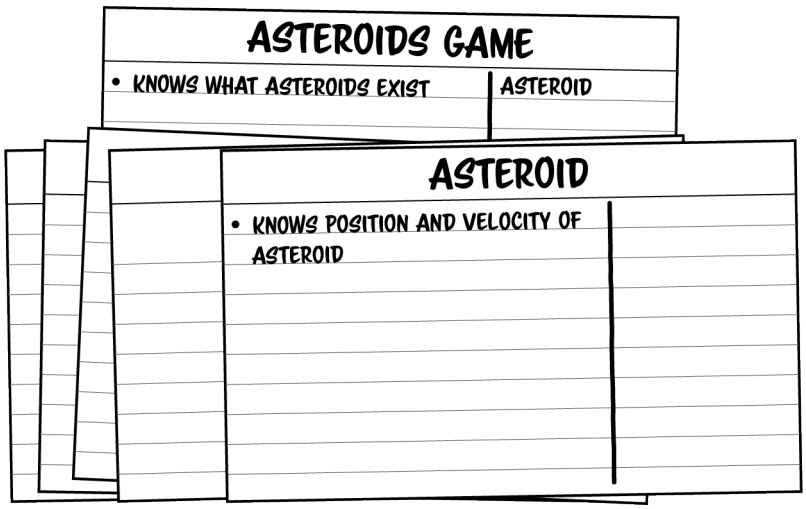
objects might work together to solve the problem. Eventually, you will discover a responsibility that no current card has listed. You must either add it to an existing card or make a new card with a new class to add it to, growing your collection of cards. As you walk through these "what if" scenarios, talking through how the objects may interact to complete the scenario, you will often find yourself pointing to cards (or picking them up and holding them) as you follow the flow of execution from object to object.

Let's walk through an example. You start by gathering your supplies: cards, pens, you, your teammates, the requirements, and any code you already have written for reference. (There are online CRC card creators as well, but I find paper or whiteboards far more flexible.) We begin with the requirement that *Asteroids drift through space at some specific velocity*. The most obvious thing here is the concept of an asteroid, so we start there. Suppose we start the game with five asteroids. We might create five cards and assign them to the **Asteroid** class.



I only wrote the responsibilities of asteroids on one card. The others would be the same. (I might even just create a single Asteroid card and remember that it could represent many.)

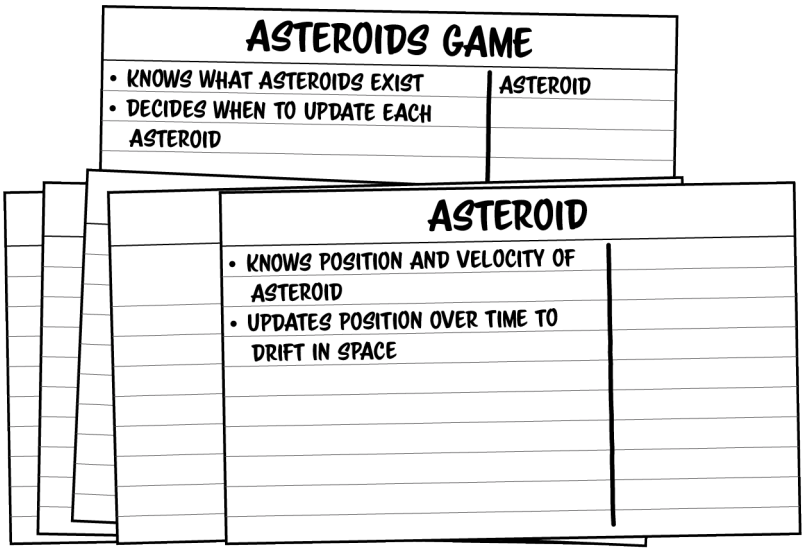
But who is keeping track of these asteroids? Who knows that these all exist? That "space" concept hints at this. These all exist within the game itself. We need a card for that:



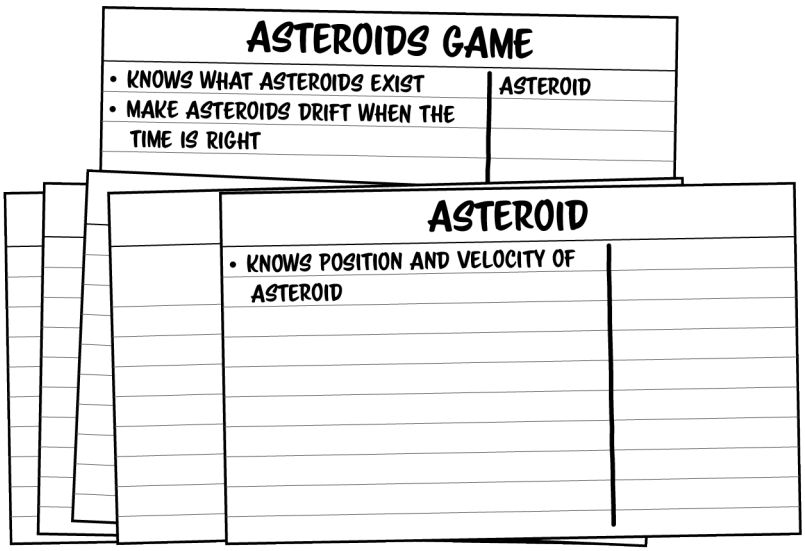
I have used the physical arrangement of these cards to reflect structure. I expected the **Asteroids Game** object to own or manage the **Asteroid** objects, so I put it above them on the table.

We still haven't addressed the actual *drifting* of asteroids yet. We have not assigned that responsibility to anybody. That responsibility needs to be given to either asteroids, the asteroid game, or a new object. This might actually be two distinct responsibilities: knowing when to update each asteroid and knowing exactly how to update each asteroid.

One approach—let's call it Option A—is to give the job of making an asteroid drift to asteroids themselves. That feels appropriate since it is changing data the asteroid owns. The responsibility of knowing *when* to update feels more at home in the **Asteroids Game** object:

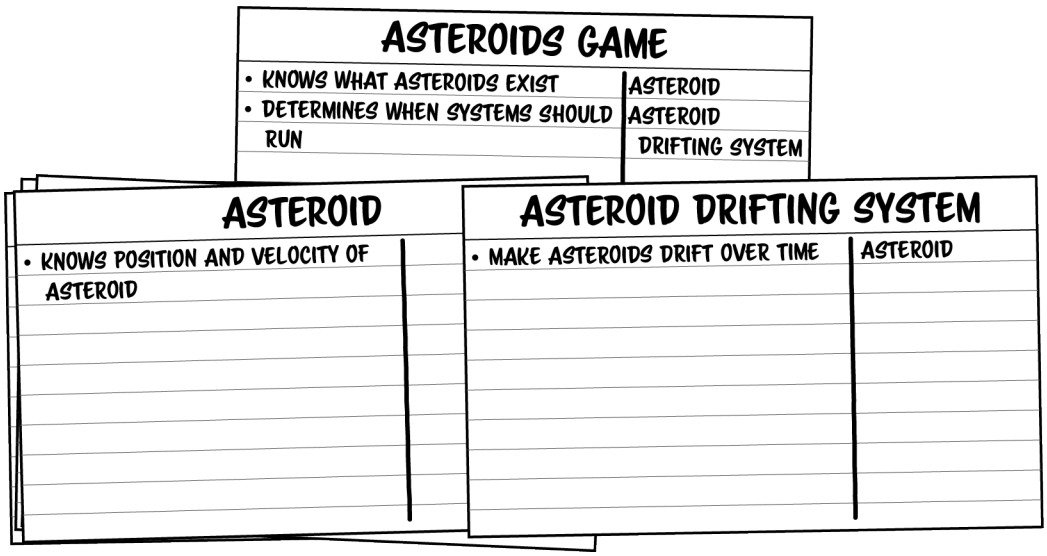


But let's consider more than one option. What else could we do? We could combine these two related responsibilities into one and just have the **Asteroids Game** object do it. This is Option B:



In this case, **Asteroids Game** would need to tell the asteroid of its new position as time passes. The **Asteroid** objects end up with just data.

Option C would be to give this responsibility to another object that doesn't exist yet. This would be an object that does nothing but update asteroid positions when the time is right. I'm going to call this the **Asteroid Drifting System** object. The **Asteroids Game** object would not update asteroid positions directly but ask this system instead. But it still owns the responsibility of knowing when the time is right:



In this case, **Asteroids Game** periodically determines that it is time to run its systems and asks the **Asteroid Drifting System** to do its job, which updates each asteroid.

At first, this may seem like a more complex solution. But consider the future under a scenario like this. We could make other systems to handle various game aspects. For example, in *Asteroids*, the player's ship eventually slows down to a stop because of drag. We could add a **Ship Drag System** object to handle that. Asteroids that reach the edge of the world wrap around to the other side. We could add a **Wraparound System** object for that. Most of the game rules could be made as a system. This approach is close to an architecture sometimes used in games called the Entity-Component-System architecture. It has some merit, but at this point, it feels more complicated than our first two options.

Evaluating a Design

In truth, we could probably make any of the above designs work, and probably several dozen other designs as well. But we do need to pick one to turn into code. How do we decide?

There are a lot of rules and guidelines that programmers will use to judge a design. We don't have time to cover them all here, but here are four of the most basic, most important rules that should give us a foundation.

Rule #1: It has to work. Look carefully at each design that you come up with. Does it do what it was supposed to do? If not, it isn't a useful design.

All three of our above options seem workable, so this rule does not eliminate anything.

Rule #2: Prefer designs that convey meaning and intent. Programmers spend more time reading code than writing it. When you come back and look at the classes, objects, and their interactions in two weeks or two years, which of the choices will be most understandable?

To shed some light on how this might work, consider this question. You have a working program handed to you (from your past self or another programmer), and you don't know how it works. But you need to make a tweak to how asteroids drift in space. Where do you look? My first thought would be to look at the **Asteroid** class. Perhaps that is a hint that having this logic live in the **Asteroid** class is better, which would give Option A an advantage.

Of the four rules, this one is the most subjective. For example, if somebody knew we were putting game rules into systems, they'd look for a drifting system. If this were the one rule not done as a system, it would be hard to remember and understand. Conveying meaning and intent is not always clear-cut.

Rule #3: Designs should not contain duplication. If one design contains the same logic or data in more than one place, it is worse than one that does not. Anything you need to change would have to be modified in many places instead of just one.

I don't think any of our options have this problem yet. But consider what things look like after adding the rule that the player's ship must also drift as asteroids do. A design that copies and pastes the drifting logic to two things is objectively worse than one that only does it once. We will learn some tools to help with that in Level 25.

Rule #4: Designs should not have unused or unnecessary elements. Make things as streamlined and straightforward as you can. Designs that add in extra stuff "just in case" are worse than ones that are as simple as possible for the current situation.

There are a few *rare* counterexamples to that rule. You should only accept a more complex design if you need the extra complexity in the immediate future. Most of the time, you can count on the fact that you can always change software later and add in the extra parts when you actually need it.

Option C might violate this rule with its additional object. That is the second time we have found an issue with Option C.

All things totaled, Option A seems like it has the most going for it, and it is what I'll turn into code next.

CREATING CODE

The next step is to turn our design into working code. Remember: creating the actual code may give us more information, and we may realize that our initial pick was not ideal. When this happens, we should adapt and change our plan. Software is soft, after all. (Have I said that enough yet?) Here is what I came up with:

```
AsteroidsGame game = new AsteroidsGame();
game.Run();

public class Asteroid
{
    public float PositionX { get; private set; }
    public float PositionY { get; private set; }
    public float VelocityX { get; private set; }
    public float VelocityY { get; private set; }
```

```
    public Asteroid(float positionX, float positionY,
                    float velocityX, float velocityY)
    {
        PositionX = positionX;
        PositionY = positionY;
        VelocityX = velocityX;
        VelocityY = velocityY;
    }

    public void Update()
    {
        PositionX += VelocityX;
        PositionY += VelocityY;
    }
}

public class AsteroidsGame
{
    private Asteroid[] _asteroids;

    public AsteroidsGame()
    {
        _asteroids = new Asteroid[5];
        _asteroids[0] = new Asteroid(100, 200, -4, -2);
        _asteroids[1] = new Asteroid(-50, 100, -1, +3);
        _asteroids[2] = new Asteroid(0, 0, 2, 1);
        _asteroids[3] = new Asteroid(400, -100, -3, -1);
        _asteroids[4] = new Asteroid(200, -300, 0, 3);
    }

    public void Run()
    {
        while (true)
        {
            foreach (Asteroid asteroid in _asteroids)
                asteroid.Update();
        }
    }
}
```

Even after making CRC cards, the act of turning something into code still requires a lot of decision-making. CRC cards don't capture every detail, just the big picture.

As you write the code, you will find other ways to improve the design. For example, those four properties on **`Asteroid`** are bothering me. Variables that begin or end the same way often indicate that you may be missing a class of some sort. We could make a **`Coordinate`** or a **`Velocity`** class with **`X`** and **`Y`** properties and simplify that to two properties. The **`X`** and **`Y`** parts are closely tied together and make more sense as a single object.

A few loose ends in this code bother me, though we don't have the tools to make it right (as I see it) yet. Here are a few that stand out to me:

- I do not like that we hardcode the starting locations of those five asteroids. We would play the same game every single time. In Level 32, we will learn about the **`Random`** class and see how it can generate random numbers for something like this.
- Array instances keep the same size once created. Right now, we are okay to have a fixed list of asteroids, but we will eventually be adding and removing asteroids from the list. In

Level 32, we will learn about the **`List`** class, which is better than arrays for changing sizes.

- My other complaint is the **`while (true)`** loop. Until we have a way to win or lose the game, looping forever is fine, but this loop updates asteroids as fast as humanly possible. (As fast as computerly possible?) One pass leads right into the next. The **`AsteroidsGame`** class has that responsibility, and it does the job; it just does it poorly. To wait a while between iterations (Level 43) or allow the asteroids to know how much time has passed and update it accordingly (Level 32) would both be improvements.

HOW TO COLLABORATE

Objects collaborate by calling members (methods, properties, etc.) on the object they need help from. Calling a method is straightforward. The tricky part is how does an object know about its collaborators in the first place? There are a variety of ways this can happen.

Creating New Objects

The first way to get a reference to an object is by creating a new instance with the **`new`** keyword. This is how the **`AsteroidsGame`** object gets a reference to the game's asteroids in the code above. These references to new **`Asteroid`** instances are put in an array and used later.

Constructor Parameters

A second way is to have something else hand it the reference when the object is created as a constructor parameter. We could have passed the asteroids to the game through its constructor like this:

```
public AsteroidsGame(Asteroid[] startingAsteroids)
{
    _asteroids = startingAsteroids;
}
```

The main method, which creates our **`AsteroidsGame`** instance, would then make the game's asteroids. Come to think of it, creating the initial list of asteroids is a responsibility we never explicitly assigned to any object. I placed the asteroid creation in **`AsteroidsGame`**, but we could have also given this responsibility to another class (maybe an **`AsteroidGenerator`** class?). Passing in the object through a constructor parameter is a popular choice if an object needs another object from the beginning but can't or shouldn't just use **`new`** to make a new one.

Method Parameters

On the other hand, if an object only needs a reference to something for a single method, it can be passed in as a method parameter.

We did not end up implementing the design that used the **`AsteroidDriftingSystem`** class. Had we done that, the game object might have given the asteroids to this object as a method parameter:

```
public class AsteroidDriftingSystem
{
    public void Update(Asteroid[] asteroids)
    {
        foreach (Asteroid asteroid in asteroids)
```

```
        {
            asteroid.PositionX += asteroid.VelocityX;
            asteroid.PositionY += asteroid.VelocityY;
        }
    }
}
```

Asking Another Object

An object can also get a reference to a collaborator by asking a third object to supply the reference. Let's say that **`AsteroidsGame`** had a public **`Asteroids`** property that returned the list of asteroids. The **`AsteroidDriftingSystem`** object could then take the game as a parameter, instead of the asteroids, and ask the game to supply the list by calling its **`Asteroids`** property:

```
public void Update(AsteroidsGame game)
{
    foreach (Asteroid asteroid in game.Asteroids)
    {
        asteroid.PositionX += asteroid.VelocityX;
        asteroid.PositionY += asteroid.VelocityY;
    }
}
```

Supplying the Reference via Property or Method

Suppose you can't supply a reference to an object in the constructor but need it for more than one method. Another option is to have the outside world supply the reference through a property or method call and then save off the reference to a field for later use. The **`AsteroidDriftingSystem`** could have done this like so:

```
public class AsteroidDriftingSystem
{
    // Initialize this to an empty array, so we know it will never be null.
    public Asteroid[] Asteroids { get; set; } = new Asteroid[0];

    public void Update()
    {
        foreach (Asteroid asteroid in Asteroids)
        {
            asteroid.PositionX += asteroid.VelocityX;
            asteroid.PositionY += asteroid.VelocityY;
        }
    }
}
```

Before this object's **`Update`** method runs, the **`AsteroidsGame`** object must ensure this property has been set. (Though it only needs to be set once, not before every **`Update`**.)

Static Members

A final approach would be to use a static property, method, or field. If it is public, these can be reached from anywhere. For example, we could make this property in **`AsteroidsGame`** to store the last created game:

```
public class AsteroidsGame
{
    public static AsteroidsGame Current { get; set; }
```

```
    // ...
}
```

When the main method runs, it can assign a value to this:

```
AsteroidsGame.Current = new AsteroidsGame();
// ...
```

Then **`AsteroidDriftingSystem`** can access the game through the static property:

```
public void Update()
{
    foreach (Asteroid asteroid in AsteroidsGame.Current.Asteroids)
    {
        asteroid.PositionX += asteroid.VelocityX;
        asteroid.PositionY += asteroid.VelocityY;
    }
}
```

In most circumstances, I recommend against this approach because it is global state (Level 21), but it has its occasional use.

Choices, Choices

You can see that there are many options for building an interconnected network of objects—almost too many. But if we make the wrong choice, we can always go back and change it.

Baby Steps

This level has been a flood of information if you are new to object-oriented programming and design. Just keep these things in mind:

You don't have to get it right the first time. It can always be changed. (Changing the structure of your code without changing what it does is called *refactoring*.)

You do not have to come up with a design to solve everything all at once. Software is typically built a little at a time, making one or several closely related requirements work before moving on to the next. Following that model makes it so that no single design cycle is too scary.

Don't be afraid to dive in and try stuff out. Your first few attempts may be rough or ugly. But if you just start trying it and seeing what is working for you and what isn't, your skills will grow quickly. (Don't worry, the whole next level will get you more practice with this stuff.)

Level 24

The Catacombs of the Class

Speedrun

This level is made entirely of problems to work through to gain more practice creating classes and doing object-oriented design and culminates in building the game of Tic-Tac-Toe from scratch.

We now know the basics of programming in C# and have more than enough skills to begin building interesting, complex programs with our knowledge. Before moving on to more advanced parts of C#, let's spend some time doing some challenges that will put our knowledge and skills to the test. This level contains nine different challenges to test your skill.

The first five challenges involve designing and programming a single class (possibly with some supporting enumerations and always with a main method that uses it).

The next three are object-oriented design challenges. *You do not need to create a working program on these.* Indeed, we haven't quite learned enough to do justice to some aspects of these challenges. (Though by the time you finish this book, you should be able to do any of these.) You only need to make an object-oriented design that you think could work in the form of CRC cards or some alternative that you feel comfortable with.

The final challenge requires you to both design and program the game of Tic-Tac-Toe. This is the most complex program we have made in our challenges. It will take some time to get it right, but that is time well spent.

Remember that you can find my answers to these challenges on the book's website.

The Five Prototypes

Narrative **Entering the Catacombs**

You arrive at the Catacombs of the Class, the place that will reveal the path to the Fountain of Objects. The Catacombs lie inside a mountain, with a wide stone entrance leading you into a series of three chambers. In the first chamber, you find five pedestals with the remnants of a class definition and specific instructions by each. Etched above a sealed doorway at the back of the room is the text, "Only the True

Programmer who can remake the Five Prototypes can proceed." Each pedestal appears to have instructions for crafting a class. These are the Five Prototypes that you must reassemble.

Boss Battle — The Point — 75 XP

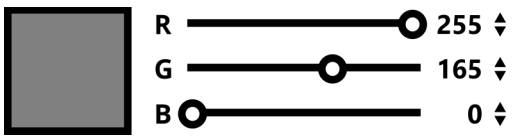
The first pedestal asks you to create a **`Point`** class to store a point in two dimensions. Each point is represented by an x-coordinate (*x*), a side-to-side distance from a special central point called the origin, and a y-coordinate (*y*), an up-and-down distance away from the origin.

Objectives:

- Define a new **`Point`** class with properties for **X** and **Y**.
- Add a constructor to create a point from a specific x- and y-coordinate.
- Add a parameterless constructor to create a point at the origin (0, 0).
- In your main method, create a point at (2, 3) and another at (-4, 0). Display these points on the console window in the format **`(x, y)`** to illustrate that the class works.
- **Answer this question:** Are your **X** and **Y** properties immutable? Why did you choose what you did?

Boss Battle — The Color — 100 XP

The second pedestal asks you to create a **`Color`** class to represent a color. The pedestal includes an etching of this diagram that illustrates its potential usage:



The color consists of three parts or channels: red, green, and blue, which indicate how much those channels are lit up. Each channel can be from 0 to 255. 0 means completely off; 255 means completely on.

The pedestal also includes some color names, with a set of numbers indicating their specific values for each channel. These are commonly used colors: White (255, 255, 255), Black (0, 0, 0), Red (255, 0, 0), Orange (255,165, 0), Yellow (255, 255, 0), Green (0, 128, 0), Blue (0, 0, 255), Purple (128, 0, 128).

Objectives:

- Define a new **`Color`** class with properties for its red, green, and blue channels.
- Add appropriate constructors that you feel make sense for creating new **`Color`** objects.
- Create static properties to define the eight commonly used colors for easy access.
- In your main method, make two **`Color`**-typed variables. Use a constructor to create a color instance and use a static property for the other. Display each of their red, green, and blue channel values.

Boss Battle — The Card — 100 XP

The digital Realms of C# have playing cards like ours but with some differences. Each card has a color (red, green, blue, yellow) and a rank (the numbers 1 through 10, followed by the symbols $, %, ^, and &). The third pedestal requires that you create a class to represent a card of this nature.

Objectives:

- Define enumerations for card colors and card ranks.
- Define a `Card` class to represent a card with a color and a rank, as described above.
- Add properties or methods that tell you if a card is a number or symbol card (the equivalent of a face card).
- Create a main method that will create a card instance for the whole deck (every color with every rank) and display each (for example, "The Red Ampersand" and "The Blue Seven").
- **Answer this question:** Why do you think we used a color enumeration here but made a color class in the previous challenge?

Boss Battle — The Locked Door — 100 XP

The fourth pedestal demands constructing a door class with a locking mechanism that requires a unique numeric code to unlock. You have done something similar before without using a class, but the locking mechanism is new. The door should only unlock if the passcode is the right one. The following statements describe how the door works.

- An open door can always be closed.
- A closed (but not locked) door can always be opened.
- A closed door can always be locked.
- A locked door can be unlocked, but a numeric passcode is needed, and the door will only unlock if the code supplied matches the door's current passcode.
- When a door is created, it must be given an initial passcode.
- Additionally, you should be able to change the passcode by supplying the current code and a new one. The passcode should only change if the correct, current code is given.

Objectives:

- Define a `Door` class that can keep track of whether it is locked, open, or closed.
- Make it so you can perform the four transitions defined above with methods.
- Build a constructor that requires the starting numeric passcode.
- Build a method that will allow you to change the passcode for an existing door by supplying the current passcode and new passcode. Only change the passcode if the current passcode is correct.
- Make your main method ask the user for a starting passcode, then create a new `Door` instance. Allow the user to attempt the four transitions described above (open, close, lock, unlock) and change the code by typing in text commands.

Boss Battle — The Password Validator — 100 XP

The fifth and final pedestal describes a class that represents a concept more abstract than the first four: a password validator. You must create a class that can determine if a password is valid (meets the rules defined for a legitimate password). The pedestal initially doesn't describe any rules, but as you brush the dust off the pedestal, it vibrates for a moment, and the following rules appear:

- Passwords must be at least 6 letters long and no more than 13 letters long.
- Passwords must contain at least one uppercase letter, one lowercase letter, and one number.

- Passwords cannot contain a capital T or an ampersand (&) because Ingelmar in IT has decreed it.

That last rule seems random, and you wonder if the pedestal is just tormenting you with obscure rules.

You ponder for a moment about how to decide if a character is uppercase, lowercase, or a number, but while scratching your head, you notice a piece of folded parchment on the ground near your feet. You pick it up, unfold it, and read it:

```
foreach with a string lets you get its characters!
> foreach (char letter in word) { ... }

char has static methods to categorize letters!
> char.IsUpper('A'), char.IsLower('a'), char.IsDigit('0')
```

That might be useful information! You are grateful to whoever left it behind. It is signed simply "A."

Objectives:

- Define a new **`PasswordValidator`** class that can be given a password and determine if the password follows the rules above.
- Make your main method loop forever, asking for a password and reporting whether the password is allowed using an instance of the **`PasswordValidator`** class.

OBJECT-ORIENTED DESIGN

Narrative — The Chamber of Design

As you finish the final class and place its complete definition back on its pedestal, the writing on each pedestal begins to glow a reddish-orange. A beam forms from each pedestal, extending upward towards the high cavernous ceiling. Additional runes on the wall begin to shine as well, and the far walls slide apart, revealing an opening further into the Catacombs.

You pass through to the next chamber and find three more pedestals with etched text. On the floor, in a ring running around the three pedestals, lie the words, "Only a True Programmer can design a system of objects for the ancient games of the people."

You must make an object-oriented design (not a complete program) for each game described on the three pedestals in the room's center to continue further.

The following three challenges will help you practice object-oriented design. **You do not need to make the full game!** You only need a starting point in the form of CRC cards (or a suitable alternative). Some parts of these games might be tough to write code for, given our current knowledge. For example, the Hangman game would be easier to read a list of words from a file, a topic covered in Level 39.

Boss Battle — Rock-Paper-Scissors — 150 XP

The first design pedestal requires you to provide an object-oriented design—a set of objects, classes, and how they interact—for the game of Rock-Paper-Scissors, described below:

- Two human players compete against each other.
- Each player picks Rock, Paper, or Scissors.

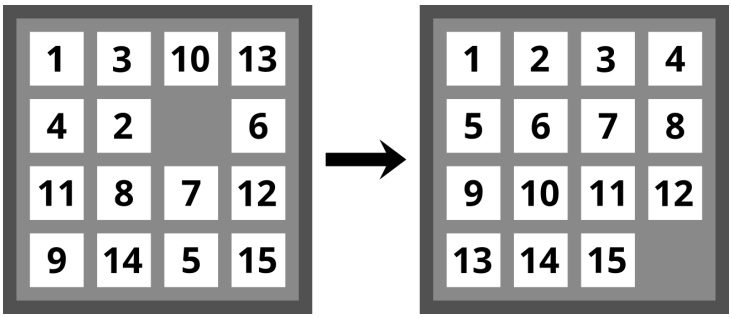
- Depending on the players' choices, a winner is determined: Rock beats Scissors, Scissors beats Paper, Paper beats Rock. If both players pick the same option, it is a draw.
- The game must display who won the round.
- The game will keep running rounds until the window is closed but must remember the historical record of how many times each player won and how many draws there were.

Objectives:

- Use CRC cards (or a suitable alternative) to outline the objects and classes that may be needed to make the game of Rock-Paper-Scissors. **You do not need to create this full game; just come up with a potential design as a starting point.**

Boss Battle — 15-Puzzle — 150 XP

The second pedestal requires you to provide an object-oriented design for the game of 15-Puzzle.



The game of 15-Puzzle contains a set of numbered tiles on a board with a single open slot. The goal is to rearrange the tiles to put the numbers in order, with the empty space in the bottom-right corner.

- The player needs to be able to manipulate the board to rearrange it.
- The current state of the game needs to be displayed to the user.
- The game needs to detect when it has been solved and tell the player they won.
- The game needs to be able to generate random puzzles to solve.
- The game needs to track and display how many moves the player has made.

Objectives:

- Use CRC cards (or a suitable alternative) to outline the objects and classes that may be needed to make the game of 15-Puzzle. **You do not need to create this full game; just come up with a potential design as a starting point.**
- **Answer this question:** Would your design need to change if we also wanted 3×3 or 5×5 boards?

Boss Battle — Hangman — 150 XP

The third pedestal in this room requires you to provide an object-oriented design for the game of Hangman. In Hangman, the computer picks a random word for the player to guess. The player then proceeds to guess the word by selecting letters from the alphabet, which get filled in, progressively revealing the word. The player can only get so many letters wrong (a letter not found in the word) before losing the game. An example run of this game could look like this:

```
Word: _ _ _ _ _ _ _ _ _ | Remaining: 5 | Incorrect:  | Guess: e
Word: _ _ _ _ _ _ _ _ E | Remaining: 5 | Incorrect:  | Guess: i
Word: I _ _ _ _ _ _ _ E | Remaining: 5 | Incorrect:  | Guess: u
Word: I _ _ U _ _ _ _ E | Remaining: 5 | Incorrect:  | Guess: o
Word: I _ _ U _ _ _ _ E | Remaining: 4 | Incorrect: O | Guess: a
Word: I _ _ U _ A _ _ E | Remaining: 4 | Incorrect: O | Guess: t
Word: I _ _ U T A _ _ E | Remaining: 4 | Incorrect: O | Guess: s
Word: I _ _ U T A _ _ E | Remaining: 3 | Incorrect: OS | Guess: r
Word: I _ _ U T A _ _ E | Remaining: 2 | Incorrect: OSR | Guess: m
Word: I M M U T A _ _ E | Remaining: 2 | Incorrect: OSR | Guess: l
Word: I M M U T A _ L E | Remaining: 2 | Incorrect: OSR | Guess: b
Word: I M M U T A B L E
You won!
```

- The game picks a word at random from a list of words.
- The game's state is displayed to the player, as shown above.
- The player can pick a letter. If they pick a letter they already chose, pick again.
- The game should update its state based on the letter the player picked.
- The game needs to detect a win for the player (all letters have been guessed).
- The game needs to detect a loss for the player (out of incorrect guesses).

Objectives:

- Use CRC cards (or a suitable alternative) to outline the objects and classes that may be needed to make the game of Hangman. **You do not need to create this full game; just come up with a potential design as a starting point.**

TIC-TAC-TOE

This final challenge requires building a more complex object-oriented program from start to finish: the game of Tic-Tac-Toe. This is the most significant program we have made so far, so expect to take some time to get it right.

Boss Battle **Tic-Tac-Toe** **300 XP**

Completing designs for the three games in the Chamber of Design causes the pedestals to light up red again, and another door opens, letting you into the final chamber. This chamber has only a single large, broad pedestal. Inscribed on the stone floor in a circle around the pedestal are the engraved words, "Only a True Programmer can build object-oriented programs."

More text engraved on the pedestal describes what you recognize as the game of Tic-Tac-Toe, stating that in ancient times, inhabitants of the land would use this as a Battle of Wits to determine the outcome of political strife. Instead of fighting wars, they would battle it out in a game of Tic-Tac-Toe.

Your job is to recreate the game of Tic-Tac-Toe, allowing two players to compete against each other. The following features are required:

- Two human players take turns entering their choice using the same keyboard.
- The players designate which square they want to play in. **Hint:** You might consider using the number pad as a guide. For example, if they enter 7, they have chosen the top left corner of the board.
- The game should prevent players from choosing squares that are already occupied. If such a move is attempted, the player should be told of the problem and given another chance.

- The game must detect when a player wins or when the board is full with no winner (draw/"cat").
- When the game is over, the outcome is displayed to the players.
- The state of the board must be displayed to the player after each play. **Hint:** One possible way to show the board could be like this:

```
It is X's turn.
   | X |
---+---+---
   | O | X
---+---+---
 O |   |
What square do you want to play in?
```

Objectives:

- Build the game of Tic-Tac-Toe as described in the requirements above. Starting with CRC cards is recommended, but the goal is to make working software, not CRC cards.
- **Answer this question:** How might you modify your completed program if running multiple rounds was a requirement (for example, a best-out-of-five series)?

Narrative The Gift of Object Sight

As you place the finished Tic-Tac-Toe program onto the pedestal, writing etched into the stone walls begins to glow reddish-orange. The glow is bright enough that you have to shield your eyes with your hand for a moment before the glowing dims to a more manageable intensity.

Suddenly, you realize that you are no longer the only thing in the room. Thousands of faintly glowing, bluish objects of various shapes and sizes float in the air around you.

You hear a resounding, booming voice echo through the chamber: "We are the Guardians of the Catacombs. We have seen your creations and know that you are a True Programmer. We have deemed you worthy of the Gift of Object Sight—the ability to see objects in code and requirements and craft solutions from objects and types.

"We need your help. The Fountain of Objects—the lifeblood of this island—has been destroyed by the vile Uncoded One. Use the Gift of Object Sight to reforge the Fountain of Objects. Without the Fountain, this land will crumble and fade into oblivion. Object Sight will lead you to the Fountain. Depart now and save this land!"

As you leave the Catacombs of the Class, you discover that your new Object Sight ability has made countless code objects visible in the world around you. You also see a distinct trail, marked with a faint blue line heading into the rugged, distant mountains where the Fountain of Objects supposedly lies. Though the journey ahead is still long, the pathway to the Fountain of Objects is now clear!

Level 25

Inheritance

Speedrun

- Inheritance lets you derive new classes based on existing ones. The new class inherits everything except constructors from the base class. `class Derived : Base { ... }`
- Classes derive from `object` by default, and everything eventually derives from `object` even if another class is explicitly stated.
- Constructors in a derived class must call out the constructor they are using from the base class unless they are using a parameterless constructor: `public Derived(int x) : base(x) { ... }`
- Derived class instances can be used where the base class is expected: `Base x = new Derived();`
- The `protected` accessibility modifier makes things accessible in the class and any derived classes.

Sometimes, a class is a subtype or specialization of another. The broader category has a set of capabilities that the subtype or specialization extends or enhances with more. Here are a few real-world examples of this type of relationship:

- Every vehicle has a top speed, maximum acceleration, passenger count, and the ability to drive it. Specific subtypes of vehicles do that and more. A truck includes a bed with the capacity to carry cargo. A tank includes a gun. Both tanks and trucks add unique information on top, but you could use a tank or a truck for anything you might do with a vehicle in a pinch.
- Writing implements let you write text or draw pictures on paper. Pencils augment this with the ability to erase, colored pencils add color, and pens add the concept of ink levels and running out of ink. But each can be used to write and draw.
- Astronomical objects all have specific properties like location and mass, which is enough to calculate gravitational pull. Stars extend that idea by including temperature and the ability to incinerate things. Planets add information like atmosphere composition and terrain details on a rocky planet.

You can define this subtype or specialization relationship in C# code using *inheritance*. Inheritance accomplishes two critical things. First, it allows you to treat the subtypes as the

more generalized type whenever necessary. Second, it allows you to consolidate what would otherwise be duplicated or copy-and-pasted code in two closely related classes.

Let's continue with the *Asteroids* example we experimented with back in Level 23. There are many types or classes of objects that drift in space. Asteroids, bullets, and the player's ship use the same mechanics to drift through space. These are distinct classes, with their own behavior, but given only the tools we have learned before now, we would have to copy and paste that drifting logic to the **`Asteroid`**, **`Bullet`**, and **`Ship`** classes as we created them.

This relationship between a subcategory and its parent category is common in object-oriented programming. A relationship where a type can expand upon another is called an *inheritance relationship*. Inheritance is our fourth key principle of object-oriented programming:

Object-Oriented Principle #4: Inheritance—Basing one class on another, retaining the original class's functionality while extending the new class with additional capabilities.

INHERITANCE AND THE OBJECT CLASS

When we define an inheritance relationship between two classes, three things happen. The new class gets everything the old class had, the new class can add in extra stuff, and the new class can always be treated as though it were the original since it has all of those capabilities.

The original class we build on is the *base class*, though it is sometimes called the *parent class* or the *superclass*. The new class that extends the base class is the *derived class*, though it is sometimes called the *child class* or the *subclass*. (Programmers aren't always great at consistent terminology.) People will say that the derived class *derives from* the base class or that the derived class *extends* the base class. Let's make that clearer with a concrete example. As it turns out, we have been unknowingly using inheritance for a while. Every class you define automatically has a base class called **`object`**. When we made an **`Asteroid`** or a **`Point`** class, these were derived from or extended the **`object`** class. **`Asteroid`** and **`Point`** are the derived classes in this relationship, and **`object`** is the base class.

The **`object`** class is special. It is the base class of everything, and everything is a specialization of the **`object`** class. That means anything the **`object`** class defines exists on every single object ever created. Let's explore the **`object`** class and get our first peek at how inheritance works.

To start, you can create instances of the **`object`** class and use **`object`** as a type for a variable:

```
object thing = new object();
```

The **`object`** class doesn't have many responsibilities, so creating instances of **`object`** itself is relatively rare. It has several methods, but we will look at two here: **`ToString`** and **`Equals`**. The **`ToString`** method creates a string representation of any object. The default implementation is to display the full name of the object's type:

```
Console.WriteLine(thing.ToString());
```

That code will display **`System.Object`**, since the **`Object`** class lives in the **`System`** namespace.

The **`Equals`** method returns a **`bool`** that indicates whether two things are considered equal or not. The following code will display **`True`** and then **`False`**.

```
object a = new object();
object b = a;
object c = new object();
Console.WriteLine(a.Equals(b));
Console.WriteLine(a.Equals(c));
```

By default, **Equals** will return whether two things are references to the same object on the heap. But equality is a complex subject in programming. Should two things be equal only if they represent the same memory location? Should they be equal if they are of the same type and their fields are equal? Do some fields matter while others do not? Under different circumstances, each of these could be true.

As we will see in the next level, your classes can sometimes redefine a method, including both **ToString** and **Equals**.

Because **object** defines the **ToString** and **Equals** methods, and because the classes we have created are derived from **object**, our objects also have **ToString** and **Equals**.

Suppose we have a simple **Point** class defined like this:

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

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

Even though this class does not define **ToString** or **Equals** methods, it has them:

```
Point p1 = new Point(2, 4);
Point p2 = p1;
Console.WriteLine(p1.ToString());
Console.Write(p1.Equals(p2));
```

That is because **Point** inherits these methods from its base class, **object**.

Importantly, because a derived class has all the base class's capabilities, you can use the derived class anywhere the based class is expected. A simple example is this:

```
object thing = new Point(2, 4);
```

The variable holds a reference to something with a type of **object**. We give it a reference to a **Point** instance. **Point** is a different class than **object**, but **Point** is derived from **object** and can thus be treated as one.

This makes things interesting. The **thing** variable knows it holds **object**s. You can use its **ToString** and **Equals** method. But the variable makes no promises that it has a reference to anything more specific than **object**:

```
Console.WriteLine(thing.ToString()); // Safe.
Console.WriteLine(thing.X);          // Compiler error.
```

Even though we put an instance of **Point** into our **thing** variable, the variable itself can only guarantee it has a reference to an **object**. It *could be* a **Point**, but the variable and the

compiler cannot guarantee that, even though a human can see it from inspecting the code. Once we place a reference to a derived class like **`Point`** into a base class variable like **`object`**, that information is not lost forever. Later in this level, we will see how we can explore an object's type and cast to the derived type if needed to regain access to the object as the derived type.

Choosing Base Classes

By default, all classes inherit from **`object`** (they use **`object`** as their base class), but it is not hard to claim a different class as the base class.

This section's code is not a complete set of useful classes, just an illustration of inheritance. In previous levels, we talked about the classes we might define for an *Asteroids* game and even made an **`Asteroid`** class once, which included the logic for drifting through space. We mentioned in passing that bullets and the player's ship would need the same behavior. We could make a **`GameObject`** class that served as a base class for all of these:

```
public class GameObject
{
    public float PositionX { get; set; }
    public float PositionY { get; set; }
    public float VelocityX { get; set; }
    public float VelocityY { get; set; }

    public void Update()
    {
        PositionX += VelocityX;
        PositionY += VelocityY;
    }
}
```

We can now create an **`Asteroid`** class that includes things specific to just the asteroid and indicate that this class is derived from **`GameObject`** instead of plain **`object`**:

```
public class Asteroid : GameObject
{
    public float Size { get; }
    public float RotationAngle { get; }
}
```

As shown above, a class identifies its base class by placing its name after a colon. **`Asteroid`** will inherit **`PositionX`**, **`PositionY`**, **`VelocityX`**, **`VelocityY`**, and **`Update`** from its base class, **`GameObject`**. It also adds new **`Size`** and **`RotationAngle`** properties, which are unique to **`Asteroid`**.

Let's suppose we also make **`Bullet`** and **`Ship`** classes that also derive from **`GameObject`**. We could set up a new game of *Asteroids* with a collection of game objects of mixed types like this:

```
GameObject[] gameObjects = new GameObject[]
{
    new Asteroid(), new Asteroid(), new Asteroid(),
    new Bullet(), new Ship()
};
```

Okay, you probably wouldn't start the game with a bullet already flying around, but you get the idea. The array stores references to **`GameObject`** instances. But that array contains

instances of the **Asteroid**, **Bullet**, and **Ship** classes. The array is fine with this because all three of those types derive from **GameObject**.

Here is where things get interesting:

```
foreach (GameObject item in gameObjects)
    item.Update();
```

Even though we are dealing with four total classes (one base class and three derived classes), we can call the **Update** method on any of them since it is defined by **GameObject**. All of the derived classes are guaranteed to have that method.

Inheritance only goes one way. While you can use an **Asteroid** when a **GameObject** is needed, you cannot use a **GameObject** where an **Asteroid** is needed. Nor can you use an **Asteroid** when a **Ship** or **Bullet** is needed:

```
Asteroid asteroid = new GameObject(); // COMPILER ERROR!
Ship ship = new Asteroid();           // COMPILER ERROR!
```

A collection of classes related through inheritance, such as these four, is called an *inheritance hierarchy*.

Inheritance hierarchies can be as deep as you need them to be. For example:

```
public class Scout : Ship { /* ... */ }

public class Bomber : Ship { /* ... */ }
```

The **Bomber** and **Scout** classes derive from **Ship**, which derives from **GameObject**, which derives from **object**. You can use a **Bomber** anywhere a **Ship**, **GameObject**, or **object** is needed.

However, classes may only choose one base class. You cannot directly derive from more than one. There are situations where this is somewhat limiting, but complications arise from this, so the C# language forbids inheriting from more than one base class.

CONSTRUCTORS

A derived class inherits most members from a base class but not constructors. Constructors put a new object into a valid starting state. A constructor in the base class can make no guarantees about the validity of an object of a derived class. So constructors are not inherited, and derived classes must supply their own. However, we can—and must—leverage the constructors defined in the base class when making new constructors in the derived class.

If a parameterless constructor exists in the base class, a constructor in a derived class will automatically call it before running its own code. And remember: if a class does not define any constructor, the compiler will generate a simple, parameterless constructor. The compiler-made one will work fine for our purposes here. This is what has happened in our simple inheritance hierarchy. Neither **GameObject** nor **Asteroid** specifically defined any constructors. The compiler generated a default parameterless constructor in both classes, and the one in **Asteroid** automatically called the one in **GameObject**.

The same thing happens if you have manually made parameterless constructors:

```
public class GameObject
{
```

```
    public GameObject()
    {
        PositionX = 2; PositionY = 4;
    }

    // Properties and other things here.
}

public class Asteroid : GameObject
{
    public Asteroid()
    {
        RotationAngle = -1;
    }

    // Properties and other things here.
}
```

Here, **Asteroid**'s parameterless constructor will automatically call **GameObject**'s parameterless constructor. Calling **new Asteroid()** will enter **Asteroid**'s constructor and immediately jump to **GameObject**'s parameterless constructor to set **PositionX** and **PositionY** and then return to **Asteroid**'s constructor to set **RotationAngle**.

Suppose a base class has more than one constructor or does not include a parameterless constructor (both common scenarios). In that case, you will need to expressly state which base class constructor to build upon for any new constructors in the derived class.

Let's suppose **GameObject** has only this constructor:

```
public GameObject(float positionX, float positionY,
                  float velocityX, float velocityY)
{
    PositionX = positionX; PositionY = positionY;
    VelocityX = velocityX; VelocityY = velocityY;
}
```

Since there is no parameterless constructor to call, any constructors defined in **Asteroid** will need to specifically indicate that it is using this other constructor and supply arguments for its parameters:

```
public Asteroid() : base(0, 0, 0, 0)
{
}
```

It is relatively common to pass along parameters from the current constructor down to the base class's constructor, so the following might be more common:

```
public Asteroid(float positionX, float positionY,
                float velocityX, float velocityY)
        : base(positionX, positionY, velocityX, velocityY)
{
}
```

(Note that I wrapped this line twice because of the limitations of the printed medium. In actual code, I might have put everything before the curly braces on a single line.)

We saw something similar in Level 18, just with the keyword **this** instead of **base**. It works in the same way, just reaching down to the base class's constructors instead of this class's constructors. You cannot use both **this** and **base** together on a constructor, but a

constructor can call out another constructor in the same class with **this** instead of using **base**. Since constructor calls with **this** cannot create a loop, eventually, something will need to pick a constructor from the base class.

Those rules are a bit complicated, so let's recap. Constructors are not inherited like other members are. Constructors in the derived class must call out a constructor from the base class (with **base**) to build upon. Alternatively, they can call out a different one in the same class (with **this**). If a parameterless constructor exists, including one the compiler generates, you do not need to state it explicitly with **base**. But don't worry; the compiler will help you spot any problems.

CASTING AND CHECKING FOR TYPES

If you ever have a base type but need to get a derived type out of it, you have some options. Consider this situation:

```
GameObject gameObject = new Asteroid();
Asteroid asteroid = gameObject; // ERROR!
```

The **gameObject** variable can only guarantee that it has a **GameObject**. It might reference something more specific, like an **Asteroid**. In the above code, we know that's true.

By casting, we can get the computer to treat the object as the more specialized type:

```
GameObject gameObject = new Asteroid();
Asteroid asteroid = (Asteroid)gameObject; // Use with caution.
```

Casting tells the compiler, "I know more about this than you do, and it will be safe to treat this as an asteroid." The compiler will allow this code to compile, but the program will crash when running if you are wrong. The above code is guaranteed to be safe, but this one is not:

```
Asteroid probablyAnAsteroid = (Asteroid)CreateAGameObject();

GameObject CreateAGameObject() { ... }
```

This cast is risky. It assumes it will get an **Asteroid** back, but that's not a guaranteed thing. If **CreateAGameObject** returns anything else, this program will crash.

Casting from a base class to a derived class is called a *downcast*. Incidentally, that is how you should feel when doing it. You should not generally do it, and usually only if you check for the correct type first. There are three ways to do this check.

The first way is with **object**'s **GetType()** method and the **typeof** keyword:

```
if (gameObject.GetType() == typeof(Asteroid)) { ... }
```

For each type that your program uses, the C# runtime will create an object representing information about that type. These objects are instances of the **Type** class, which is a type that has metadata about other types in your program. Calling **GetType()** returns the type object associated with the instance's class. If **gameObject** is an **Asteroid**, it will return the **Type** object representing the **Asteroid** class. If it is a **Ship**, **GetType** will return the **Type** object representing the **Ship** class. The **typeof** keyword lets you access these special objects by name instead. Using code like this, you can see if an object's type matches some specific class.

Using **typeof** and **.GetType()** only work if there is an exact match. If you have an **Asteroid** instance and do **asteroid.GetType() == typeof(GameObject)**, this evaluates to **false**. The **Type** instances that represent the **Asteroid** and **GameObject** classes are different. That can work for or against you, but it is important to keep in mind.

Another way is through the **as** keyword:

```
GameObject gameObject = CreateAGameObject();
Asteroid? asteroid = gameObject as Asteroid;
```

The **as** keyword simultaneously does a check *and* the conversion. If **gameObject** is an **Asteroid** (or something derived from **Asteroid**), then the variable **asteroid** will contain the reference to the object, now known to be an **Asteroid**. If **gameObject** is a **Ship** or a **Bullet**, then **asteroid** will be **null**. That means you will want to do a null check before using the variable.

The third way is with the **is** keyword. The **is** keyword is powerful and is one way to use patterns, which is the topic of Level 40. But it is frequently used to simply check the type and assign it to a new variable. The most common way to use it is like this:

```
if (gameObject is Asteroid asteroid)
{
    // You can use the `asteroid` variable here.
}
```

If you don't need the variable that this creates, you can skip the name:

```
if (gameObject is Asteroid) { ... }
```

THE PROTECTED ACCESS MODIFIER

We have encountered three accessibility modifiers in the past: **private**, **public**, and **internal**. The fourth accessibility modifier is the **protected** keyword. If something is protected, it is accessible within the class and also any derived classes. For example:

```
public class GameObject
{
    public float PositionX { get; protected set; }
    public float PositionY { get; protected set; }
    public float VelocityX { get; protected set; }
    public float VelocityY { get; protected set; }
}
```

If we make these setters **protected** instead of **public**, only **GameObject** and its derived classes (like **Asteroid** and **Ship**) can change those properties; the outside world cannot.

SEALED CLASSES

If you want to forbid others from deriving from a specific class, you can prevent it by adding the **sealed** modifier to the class definition:

```
public sealed class Asteroid : GameObject
{
    // ...
}
```

In this case, nobody will be able to derive a new class based on **`Asteroid`**. It is rare to want an outright ban on deriving from a class, but it has its occasional uses. Sealing a class can also sometimes result in a performance boost.

Challenge — Packing Inventory — 150 XP

You know you have a long, dangerous journey ahead of you to travel to and repair the Fountain of Objects. You decide to build some classes and objects to manage your inventory to prepare for the trip.

You decide to create a **`Pack`** class to help in holding your items. Each pack has three limits: the total number of items it can hold, the weight it can carry, and the volume it can hold. Each item has a weight and volume, and you must not overload a pack by adding too many items, too much weight, or too much volume.

There are many item types that you might add to your inventory, each their own class in the inventory system. (1) An arrow has a weight of 0.1 and a volume of 0.05. (2) A bow has a weight of 1 and a volume of 4. (3) Rope has a weight of 1 and a volume of 1.5. (4) Water has a weight of 2 and a volume of 3. (5) Food rations have a weight of 1 and a volume of 0.5. (6) A sword has a weight of 5 and a volume of 3.

Objectives:

- Create an **`InventoryItem`** class that represents any of the different item types. This class must represent the item's weight and volume, which it needs at creation time (constructor).
- Create derived classes for each of the types of items above. Each class should pass the correct weight and volume to the base class constructor but should be creatable themselves with a parameterless constructor (for example, **`new Rope()`** or **`new Sword()`**).
- Build a **`Pack`** class that can store an array of items. The total number of items, the maximum weight, and the maximum volume are provided at creation time and cannot change afterward.
- Make a **`public bool Add(InventoryItem item)`** method to **`Pack`** that allows you to add items of any type to the pack's contents. This method should fail (return **`false`** and not modify the pack's fields) if adding the item would cause it to exceed the pack's item, weight, or volume limit.
- Add properties to **`Pack`** that allow it to report the current item count, weight, and volume, and the limits of each.
- Create a program that creates a new pack and then allow the user to add (or attempt to add) items chosen from a menu.

Level 26

Polymorphism

Speedrun

- Polymorphism lets a derived class supply its own definition ("override") for a member declared in its base class.
- Marking a member with **`virtual`** indicates it can be overridden.
- Derived classes override a member by marking it with **`override`**.
- Classes can leave members unimplemented with **`abstract`**, but the class must also be **`abstract`**.

Inheritance is powerful, but it is made whole with the topic of this level: *polymorphism*.

Imagine programming the game of chess. We could define **`Pawn`**, **`Rook`**, and **`King`** classes, all derived from a **`ChessPiece`** base class using inheritance. But this does not allow us to solve a fundamental problem in chess: deciding whether some move is legal or not. Each piece has different rules for determining if a move is legal or not. There is some overlap—no piece can stay put and count it as a move, and no piece can move off the 8×8 board. But beyond that, each piece is different. With just inheritance, the best we could do looks like this:

```
public class ChessPiece
{
    public int Row { get; set; }
    public int Column { get; set; }

    public bool IsLegalMove(int row, int column) =>
                IsOnBoard(row, column) && !IsCurrentLocation(row, column);

    protected bool IsOnBoard(int row, int column) =>
                row >= 0 && row < 8 && column >= 0 && column < 8;

    protected bool IsCurrentLocation(int row, int column) =>
                row == Row && column == Column;
}
```

This base class does some basic checks that make sense for all chess pieces but can go no further.

A derived class can do this:

```
public class King : ChessPiece
{
    public bool IsLegalKingMove(int row, int column)
    {
        if (!IsLegalMove(row, column)) return false;

        // Moving more than one row or one column is not a legal king move.
        if (Math.Abs(row - Row) > 1) return false;
        if (Math.Abs(column - Column) > 1) return false;

        return true;
    }
}
```

`King` adds an **`IsLegalKingMove`** method. You could imagine a similar **`IsLegalPawnMove`** in the **`Pawn`** class and so on.

Unfortunately, we would need to remember which objects are of which types to call the appropriate **`IsLegalSomethingMove`** methods.

Polymorphism allows us to solve this problem elegantly. Polymorphism means "many forms" (from Greek). It is a mechanism that lets different classes related by inheritance provide their own definition for a method. When something calls the method, the version that belongs to the object's actual type will be determined and called. Polymorphism is our fifth and final principle of object-oriented programming.

Object-Oriented Principle #5: Polymorphism—Derived classes can override methods from the base class. The correct version is determined at runtime, so you will get different behavior depending on the object's class.

In our chess example, each derived class will be able to supply its own version of **`IsLegalMove`**. When the program runs, the correct **`IsLegalMove`** method is called, depending on the actual object involved:

```
ChessPiece p1 = new Pawn();
ChessPiece p2 = new King();

Console.WriteLine(p1.IsLegalMove(2, 2));
Console.WriteLine(p2.IsLegalMove(2, 2));
```

Even though **`p1`** and **`p2`** both have the type **`ChessPiece`**, calling **`IsLegalMove`** will use the piece-specific version on the last two lines because of polymorphism.

Not every method can leverage polymorphism. A method must indicate it is allowed by placing the **`virtual`** keyword on it, giving permission to derived classes to replace it.

```
public virtual bool IsLegalMove(int row, int column) =>
                        IsOnBoard(row, column) &&
                        !IsCurrentLocation(row, column);
```

We can replace or *override* the method with an alternative version in a derived class. We could put this in the **`King`** class:

```
public override bool IsLegalMove(int row, int column)
{
    if (!base.IsLegalMove(row, column)) return false;

    // Moving more than one row or one column is not a legal king move.
```

```
    if (Math.Abs(row - Row) > 1) return false;
    if (Math.Abs(column - Column) > 1) return false;

    return true;
}
```

The **King** class has now provided its own definition for **IsLegalMove**. It has overridden the version supplied by the base class. **Pawn**, **Rook**, and the others can do so as well.

When you override a method, it is a total replacement. If you want to reuse the overridden logic from the base class, you can call it through the **base** keyword. The code above does this to keep the logic for staying on the board. Not all overrides call the base class's version of the method, but it is common.

You can override most types of members except fields and constructors (which aren't inherited anyway).

Just because a method is virtual does not mean a derived class *must* override it. With our chess example, they all probably will. In other situations, some derived classes will find the base class version sufficient.

When a normal (non-virtual) member is called, the compiler can determine which method to call at compile time. When a method is virtual, it cannot. Instead, it records some metadata in the compiled code to know what to look up as it is running. This lookup as the program is running takes a tiny bit of time. You do not want to just make everything virtual "just in case." Instead, consider what a derived class may need to replace and make only those virtual.

The overriding method must match the name and parameters (both count and type) as the overridden method. However, you can use a more specific type for the return value if you want. For example, if you have a **public virtual object Clone()** method, it can be overridden with a **public override SpecificClass Clone()** since **SpecificClass** is derived from **object**.

Abstract Methods and Classes

Sometimes, a base class wants to require that all derived classes supply a definition for a method but can't provide its own implementation. In such cases, it can define an *abstract method*, specifying the method's signature without providing a body or implementation for the method. When a class has an abstract method, derived classes *must* override the method; there is nothing to fall back on. In fact, any class with an abstract method is an incomplete class. You cannot create instances of it (only derived classes), and you must mark the class itself as abstract as well. To illustrate, here is what the **ChessPiece** class might look like with an abstract **IsLegalMove** method:

```
public abstract class ChessPiece
{
    public abstract bool IsLegalMove(int targetRow, int targetColumn);

    // ...
}
```

Adding the **abstract** keyword (instead of **virtual**) to a method says, "Not only can you override this method, but you *must* override this method because I'm not supplying a definition." Instead of a body, an abstract method ends with a semicolon. Once a class has any abstract members, the class must also be made **abstract**, as shown above.

Abstract members can only live in abstract classes, but an abstract class can contain any member it wants—abstract, virtual, or normal. It is not unheard of to have an abstract class with no abstract members—just a foundation for closely related types to build on.

When a distinction is needed, non-abstract classes are often referred to as *concrete classes*.

New Methods

If a derived class defines a member whose name matches something in a base class without overriding it, a new member will be created, which hides (instead of overrides) the base class member. This is nearly always an accident caused by forgetting the **`override`** keyword. The compiler assumes as much and gives you a warning for it.

In the rare cases where this was by design, you can tell the compiler it was intentional by adding the **`new`** keyword to the member in the derived class:

```
public class Base
{
    public int Method() => 0;
}

public class Derived : Base
{
    public new int Method() => 4;
}
```

When a new member is defined, unlike polymorphism, the behavior depends on the type of the variable involved, not the instance's type:

```
Derived d = new Derived();
Base b = d;

Console.WriteLine(d.Method() + " " + b.Method());
```

This displays **`4 0`**, not **`4 4`**, as we would otherwise expect with polymorphism.

Challenge — Labeling Inventory — 50 XP

You realize that your inventory items are not easy to sort through. If you could make it easy to label all of your inventory items, it would be easier to know what items you have in your pack.

Modify your inventory program from the previous level as described below.

Objectives:

- Override the existing **`ToString`** method (from the **`object`** base class) on all of your inventory item subclasses to give them a name. For example, **`new Rope().ToString()`** should return **`"Rope"`**.
- Override **`ToString`** on the **`Pack`** class to display the contents of the pack. If a pack contains water, rope, and two arrows, then calling **`ToString`** on that **`Pack`** object could look like **`"Pack containing Water Rope Arrow Arrow"`**.
- Before the user chooses the next item to add, display the pack's current contents via its new **`ToString`** method.

Challenge The Old Robot 200 XP

You spot something shiny, half-buried in the mud. You pull it out and realize that it seems to be some mechanical automaton with the words "Property of Dynamak" etched into it. As you knock off the caked-on mud, you realize that it seems like this old automaton might even be *programmable* if you can give it the proper commands. The automaton seems to be structured like this:

```
public class Robot
{
    public int X { get; set; }
    public int Y { get; set; }
    public bool IsPowered { get; set; }
    public RobotCommand?[] Commands { get; } = new RobotCommand?[3];
    public void Run()
    {
        foreach (RobotCommand? command in Commands)
        {
            command?.Run(this);
            Console.WriteLine($"[{X} {Y} {IsPowered}]");
        }
    }
}
```

You don't see a definition of that **RobotCommand** class. Still, you think you might be able to recreate it (a class with only an abstract **Run** command) and then make derived classes that extend **RobotCommand** that move it in each of the four directions and power it on and off. (You wish you could manufacture a whole army of these!)

Objectives:

- Copy the code above into a new project.
- Create a **RobotCommand** class with a public and abstract **void Run(Robot robot)** method. (The code above should compile after this step.)
- Make **OnCommand** and **OffCommand** classes that inherit from **RobotCommand** and turn the robot on or off by overriding the **Run** method.
- Make a **NorthCommand**, **SouthCommand**, **WestCommand**, and **EastCommand** that move the robot 1 unit in the +Y direction, 1 unit in the -Y direction, 1 unit in the -X direction, and 1 unit in the +X direction, respectively. Also, ensure that these commands only work if the robot's **IsPowered** property is **true**.
- Make your main method able to collect three commands from the console window. Generate new **RobotCommand** objects based on the text entered. After filling the robot's command set with these new **RobotCommand** objects, use the robot's **Run** method to execute them. For example:

```
on
north
west

[0 0 True]
[0 1 True]
[-1 1 True]
```

- **Note:** You might find this strategy for making commands that update other objects useful in some of the larger challenges in the coming levels.

Level 27

Interfaces

Speedrun

- An interface is a type that defines a contract or role that objects can fulfill or implement: `public interface ILevelBuilder { Level BuildLevel(int levelNumber); }`
- Classes can implement interfaces: `public class LevelBuilder : ILevelBuilder { public Level BuildLevel(int levelNumber) => new Level(); }`
- A class can have only one base class but can implement many interfaces.

We've learned how to create new types using enumerations and classes, but you can make several other flavors of type definitions in C#. The next one we'll learn about is an *interface*. An interface is a type that defines an object's interface or boundary by listing the methods, properties, etc., that an object must have without supplying any behavior for them. You could also think of an interface as defining a specific role or responsibility in the system without providing the code to make it happen.

We see interfaces in the real world all the time. For example, a piano with its 88 black and white keys and an expectation that pushing the keys will play certain pitches is an interface. Electric keyboards, upright pianos, grand pianos, and in no small degree, even organs and harpsichords provide the same interface. A user of the interface—a pianist—can play any of these instruments in the same way without worrying about how they each produce sound. We see a similar thing with vehicles, which all present a steering wheel, an accelerator, and a brake pedal. As a driver, it doesn't matter if the engine is gas, diesel, or electric or whether the brakes are frictional or electromagnetic.

Interfaces give us the most flexibility in how something accomplishes its job. It is almost as though we have made a class where every member is abstract, though it is even more flexible than that.

Interfaces are perfect for situations where we know we may want to substitute entirely different or unrelated objects to fulfill a specific role or responsibility in our system. They give us the most flexibility in evolving our code over time. The only assumption made about the object is that it complies with the defined interface. As long as two things implement the same interface, we can swap one for another, and the rest of the system is none the wiser.

Defining Interfaces

Let's say we have a game where the player advances through levels, one at a time. We'll keep it simple and say that each level is a grid of different terrain types from this enumeration:

```
public enum TerrainType { Desert, Forests, Mountains, Pastures, Fields, Hills }
```

Each level is a 2D grid of these terrain types, represented by an instance of this class:

```
public class Level
{
    public int Width { get; }
    public int Height { get; }
    public TerrainType GetTerrainAt(int row, int column) { /* ... */ }
}
```

We find a use for interfaces when deciding where level definitions come from. There are many options. We could define them directly in code, setting terrain types at each row and column in C# code. We could load them from files on our computer. We could load them from a database. We could randomly generate them. There are many options, and each possibility has the same result and the same job, role, or responsibility: create a level to play. Yet, the code for each is entirely unrelated to the code for the other options.

We may not know yet which of these we will end up using. Or perhaps we plan to retrieve the levels from the Internet but don't intend to get a web server running for a few more months and need a short-term fix.

To preserve as much flexibility as possible around this decision, we simply define what this role must do—what interface or boundary the object or objects fulfilling this role will have:

```
public interface ILevelBuilder
{
    Level BuildLevel(int levelNumber);
}
```

Interface types are defined similarly to a class but with a few differences.

For starters, you use the **`interface`** keyword instead of the **`class`** keyword.

Second, you can see that I started my interface name with a capital **`I`**. That is not strictly necessary, but it is a common convention in C# Land. Most C# programmers do it, so I recommend it as well. (It does lead to the occasional awkward double **`I`** names like **`IImmutableList`**, but you get used to it.)

Members of an interface are **`public`** and **`abstract`** automatically. After all, an interface defines a boundary (**`abstract`**) meant for others to use (**`public`**). You can place those keywords on there if you'd like, but few developers do.

Because an interface defines just the boundary or job to be done, its members don't have an implementation. (There is an exception to that statement, described later in this level.) Most things you might place in a class can also be placed in an interface (without an implementation) except fields.

While this **`ILevelBuilder`** interface only has a single method, interfaces can have as many members as they need to define the role they represent. For example, you could let the rest of the game know how many levels are in the set by adding an **`int Count { get; }`** property.

Implementing Interfaces

Once an interface has been created, the next step is to build a class that fulfills the job described by the interface. This is called *implementing the interface*. It looks like inheritance, so some programmers also call it *extending* the interface or *deriving from* the interface. These names are all common, and many C# programmers don't strongly differentiate interfaces from base classes and use the terms interchangeably. I will refer to it as implementing an interface and extending a base class in this book.

The simplest implementation of the **ILevelBuilder** interface is probably defining levels in code:

```
public class FixedLevelBuilder : ILevelBuilder
{
    public Level BuildLevel(int levelNumber)
    {
        Level level = new Level(10, 10, TerrainType.Forests);

        level.SetTerrainAt(2, 4, TerrainType.Mountains);
        level.SetTerrainAt(2, 5, TerrainType.Mountains);
        level.SetTerrainAt(6, 1, TerrainType.Desert);

        return level;
    }
}
```

The body of **BuildLevel** takes quite a few liberties that we never fleshed out. It uses a constructor and a **SetTerrainAt** method that we did not define earlier in the **Level** class, though you could imagine including them. It also creates the same level every time, ignoring the **levelNumber** parameter. In a real-world situation, we'd probably need to do more. But the vital part of that code is how **FixedLevelBuilder** implements the **ILevelBuilder** interface.

Like extending a base class through inheritance, you place a colon after the class name, followed by the interface name you are implementing.

You must define each member included in the interface, as we did with the **BuildLevel** method. These will be **public** but do not put the **override** keyword on them. This isn't an override. It is simply filling in the definition of how this class performs the job it has claimed to do by implementing the interface.

A class that implements an interface can have other members unrelated to the interfaces it implements. By indicating that a class implements an interface, you are saying that it will have *at least* the capabilities defined by the interface, not that it is limited to the interface. One notable example is that an interface can declare a property with a **get** accessor, while a class that implements it can *also* include a **set** or **init** accessor.

You can probably imagine creating other classes that implement this interface by loading definitions from files (Level 39), generating them randomly (perhaps using the **Random** class described in Level 32), or retrieving the levels from a database or the Internet.

We can create variables that use an interface as their type and place in it anything that implements that interface:

```
ILevelBuilder levelBuilder = LocateLevelBuilder();

int currentLevel = 1;
```

```
while (true)
{
    Level level = levelBuilder.BuildLevel(currentLevel);
    RunLevel(level);
    currentLevel++;
}
```

The rest of the game doesn't care which implementation of **`ILevelBuilder`** is being used. However, with the code we have written so far, we know it will be **`FixedLevelBuilder`** since that is the only one that exists. However, by doing nothing more than adding a new class that implements **`ILevelBuilder`** and changing the implementation of **`LocateLevelBuilder`** to return that instead, we can completely change the source of levels in our game. The entire rest of the game does not care where they come from, as long as the object building them conforms to the **`ILevelBuilder`** interface. We have reserved a great degree of flexibility for the future by merely defining and using an interface.

Interfaces and Base Classes

Interfaces and base classes can play nicely together. A class can simultaneously extend a base class and implement an interface. Do this by listing the base class followed by the interface after the colon, separated by commas:

```
public class MySqlDatabaseLevelBuilder : BasicDatabaseLevelBuilder, ILevelBuilder
{ ... }
```

A class can implement several interfaces in the same way by listing each one, separated by commas:

```
public class SomeClass : ISomeInterface1, ISomeInterface2 { ... }
```

While you can only have one base class, a class can implement as many interfaces as you want. (Though implementing many interfaces may signify that an object or class is trying to do too much.)

Finally, an interface itself can list other interfaces (but not classes) that it augments or extends:

```
public interface IBroaderInterface : INarrowerInterface { ... }
```

When a class implements **`IBroaderInterface`**, they will also be on the hook to implement **`INarrowerInterface`**.

Explicit Interface Implementations

Occasionally, a class implements two different interfaces containing members with the same name but different meanings. For example:

```
public interface IBomb { void BlowUp(); }
public interface IBalloon { void BlowUp(); }

public class ExplodingBalloon : IBomb, IBalloon
{
    public void BlowUp() { ... }
}
```

This single method is enough to implement both **IBomb** and **IBalloon**. If this one method definition is a good fit for both interfaces, you are done.

On the other hand, in this situation, "blow up" means different things for bombs than it does balloons. When we define **ExplodingBalloon**'s **BlowUp** method, which one are we referring to?

If you have control over these interfaces, consider renaming one or the other. We could rename **IBomb.BlowUp** to **Detonate** or **IBalloon.BlowUp** to **Inflate**. Problem solved.

But if you don't want to or can't, the other choice is to make a definition for each using an *explicit interface implementation*:

```
public class ExplodingBalloon : IBomb, IBalloon
{
    void IBomb.BlowUp() { Console.WriteLine("Kaboom!"); }
    void IBalloon.BlowUp() { Console.WriteLine("Whoosh"); }
}
```

By prefacing the method name with the interface name, you can define two versions of **BlowUp**, one for each interface. Note that the **public** has been removed. This is required with explicit interface implementations.

The big surprise is that explicit implementations are detached from their containing class:

```
ExplodingBalloon explodingBalloon = new ExplodingBalloon();
// explodingBalloon.BlowUp(); // COMPILER ERROR!

IBomb bomb = explodingBalloon;
bomb.BlowUp();

IBalloon balloon = explodingBalloon;
balloon.BlowUp();
```

In this situation, you cannot call **BlowUp** directly on **ExplodingBalloon**! Instead, you must store it in a variable that is either **IBomb** or **IBalloon** (or cast it to one or the other). Then it becomes available because it is no longer ambiguous.

If one of the two is more natural for the class, you can choose to do an explicit implementation for only one, leaving the other as the default. If you do this, then the non-explicit implementation is accessible on the object without casting it to the interface type.

DEFAULT INTERFACE METHODS

Interfaces allow you to create a default implementation for methods with some restrictions. (If you do not like these restrictions, an abstract base class may be a better fit.) Default implementations are primarily for growing or expanding an existing interface to do more. Imagine having an interface with ten classes that implement the interface. If you want to add a new method or property to this interface, you have to revisit each of the ten implementations to adapt them to the new changes.

If you can get an interface definition right the first time around, it saves you from this rework. It is worth taking time to try to get interfaces right, but we can never guarantee that. Sometimes, things just need to change.

Of course, you can just go for it and add the new member to each of the many implementations. This is often a good, clean solution, even though it takes time.

In other situations, providing a default implementation for a method can be a decent alternative. Imagine you have an interface that a thousand programmers around the world use. If you change the interface, they'll all need to update their code. A default implementation may save a lot of pain for many people.

Let's suppose we started with this interface definition:

```
public interface ILevelBuilder
{
    Level BuildLevel(int levelNumber);
    int Count { get; }
}
```

If we wanted to build all the levels at once, we might consider adding a **`Level[] BuildAllLevels()`** method to this interface. Adding this would not be complicated:

```
public interface ILevelBuilder
{
    Level BuildLevel(int levelNumber);
    int Count { get; }
    Level[] BuildAllLevels();
}
```

But the logic for this is pretty standard, and if we can just make a default implementation for **`BuildAllLevels`**, nobody is required to make their own. We can grow the interface almost for free:

```
public interface ILevelBuilder
{
    Level BuildLevel(int levelNumber);
    int Count { get; }

    Level[] BuildAllLevels()
    {
        Level[] levels = new Level[Count];

        for (int index = 1; index <= Count; index++)
            levels[index - 1] = BuildLevel(index);

        return levels;
    }
}
```

With this default implementation, nobody else will have to write a **`BuildAllLevels`** method unless they need something special. But if they do, it is a simple matter of adding a definition for the method in the class.

A default implementation can use the other members of the interface. We see that above since **`BuildAllLevels`** calls both **`Count`** and **`BuildLevel`**.

Supporting Default Interface Methods

Default interface method implementations are a relatively new thing to C#. When they decided to add this feature, they provided many of the tools needed to do it well. For example, if a single method becomes too big, you can split some of the code into private methods. You

can also create protected methods and static methods. I won't get into all the details because default method implementations are not all that common, and the compiler will tell you if you attempt something that does not work. However, one significant constraint is that you cannot add instance fields. Interfaces cannot contain data themselves. (Though static fields are allowed.)

Should I Use Default Interface Methods?

Adding default implementations in an interface was a somewhat controversial change. It is hard for those making widely used interfaces to update every implementing class. The benefits of default implementations are a lifesaver to them. But for many others, the benefits are small, and it serves little value other than to cloud the concept of an interface.

Should you embrace them and provide one for every method you make, avoid them like the plague, or something in between?

My recommendation stems from the fact that interfaces are meant to define just the boundary, not the implementation. I suggest skipping default implementations except when many classes implement the interface and when the default implementation is nearly always correct for the classes that use the interface.

Not every interface change can be solved with default method implementations. It only works if you are adding new stuff to an interface. If you are renaming or removing a method, you will just need to fix all the classes that implement the interface.

Challenge **Robotic Interface** **75 XP**

With your knowledge of interfaces, you realize you can refine the old robot you found in the mud to use interfaces instead of the original design. Instead of an abstract **`RobotCommand`** base class, it could become an **`IRobotCommand`** interface!

Building on your solution to the Old Robot challenge, perform the changes below:

Objectives:

- Change your abstract **`RobotCommand`** class into an **`IRobotCommand`** interface.
- Remove the unnecessary **`public`** and **`abstract`** keywords from the **`Run`** method.
- Change the **`Robot`** class to use **`IRobotCommand`** instead of **`RobotCommand`**.
- Make all of your commands implement this new interface instead of extending the **`RobotCommand`** class that no longer exists. You will also want to remove the **`override`** keywords in these classes.
- Ensure your program still compiles and runs.
- **Answer this question:** Do you feel this is an improvement over using an abstract base class? Why or why not?

Level 28

Structs

Speedrun

- A struct is a custom value type that defines a data structure without complex behavior: **`public struct Point { ... }`**. Structs are not focused on behavior but can have properties and methods.
- Compared to classes: structs are value types, automatically have value semantics, and cannot be used in inheritance.
- Make structs small, immutable, and ensure the default value is legitimate.
- All the built-in types are aliases for other structs (and a few classes). For example, **`int`** is shorthand for **`System.Int32`**.
- Value types can be stored in reference-typed variables (**`object thing = 3;`**) but will cause the value to be boxed and placed on the heap.

While classes are a great way to create new reference types, C# also lets you make custom value types. New types of this nature are called *structs*, which is short for *structure* or *data structure*.

Making a struct is nearly the same as making a class. We have seen many variations on a **`Point`** class before, but here is a **`Point`** struct:

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

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

The only code difference is using the **`struct`** keyword instead of the **`class`** keyword. Most aspects of making a struct are the same as making a class. You can add fields, properties, methods, and constructors, along with some other member types we have not discussed yet).

Using a struct is also nearly the same as using a class:

```
Point p1 = new Point(2, 4);
Console.WriteLine($"({p1.X}, {p1.Y}");
```

The critical difference is that structs are value types instead of reference types. That means variables whose type is a struct contain the data where the variable lives, instead of holding a reference that points to the data, as is the case with classes.

Recall that the variable's contents are copied when passing something between methods (an argument or return value). For a reference type like classes, that means the reference is copied. The calling method and the called method both have their own reference, but both references point to the same object. For a value type like structs, the entire block of data is copied, and each ends up with a full copy of the data. The same is true when assigning a value from one local variable to another or working with expressions.

Structs are primarily useful for representing small data-related concepts that don't have a lot of behavior. Representing a 2D point, as we did above, is a good candidate for a struct. A circle, a line, or a matrix could also be good candidates. In situations where the concept is not a small data-related concept, a class is usually better. Even still, some small data-related concepts are still better as a class. We'll analyze the class vs. struct decision in more depth in a moment.

MEMORY AND CONSTRUCTORS

Because structs are value types, memory usage and constructors are two critical ways structs differ from the classes we have been making in the past.

Reference types, such as a class, can be null (Level 22). In these cases, the memory for an object doesn't exist until it is explicitly created by calling a constructor with the **`new`** keyword. For value types like structs, we don't have that option. The variable's mere existence means its memory must also exist, even before it has been initialized by a constructor. This model has a lot of implications that may be surprising.

First, while a constructor can be used to initialize data, invoking a constructor is not always necessary. Consider this struct:

```
public struct PairOfInts
{
    public int A; // These are public fields, which are usually best to avoid.
    public int B;
}
```

Now, look at this code with a **`PairOfInts`** local variable:

```
PairOfInts pair;
pair.A = 2;
Console.WriteLine(pair.A);
```

It calls no constructor but still assigns a value to its **`A`** field. The **`pair`** variable acts like two separate local variables, each of which can be initialized and used like any other local variable but through a shared name.

Now imagine we add this class into the mix:

```
public class PairOfPairs
{
    public PairOfInts _one;
    public PairOfInts _two;
```

```
    public void Display()
    {
        Console.WriteLine($"{_one.A} {_one.B} and {_two.A} {_two.B}");
    }
}
```

Once again, we can use these structs without calling a constructor. In this case, the structs are initialized to default values by zeroing out their memory, meaning **`A`** and **`B`** of both **`_one`** and **`_two`** will be **`0`** until somebody changes it.

No matter what constructors you give a struct, they may simply not be called!

Second, structs will always have a public parameterless constructor. If a class doesn't define any constructors, the compiler automatically generates a parameterless constructor for any class you make. The compiler does the same thing for a struct. For a class, if you define a different constructor, the compiler no longer makes a parameterless constructor. For a struct, the compiler will define a public parameterless constructor anyway. You cannot get rid of the public parameterless constructor. However, you may define this public, parameterless constructor yourself if you need it to do something specific.

Third, field initializers are a bit weird in a struct. Consider this version of **`PairOfInts`**:

```
public struct PairOfInts
{
    public int A = 10;
    public int B = -2;
}
```

These initializers do not always run when you use a **`PairOfInts`**. More specifically:

- Field and property initializers don't ever run if no constructor is called.
- The compiler-generated constructor runs these initializers only if the struct has no constructors.
- If you add your own constructors, these initializers will only run as a part of constructors *you* have defined, not as part of the compiler-generated one.

To ensure the third rule doesn't catch you off guard, you will likely want to define your own parameterless constructor when adding initializers to your fields or properties.

You don't need to memorize all these rules. Just remember that it can be a tricky area. Don't just assume your code works, but check to ensure it does.

Classes vs. Structs

Classes and structs have a lot in common, but let's take some time to compare the two and describe when you might want each.

The main difference is that classes are reference types and structs are value types. We touched on this in the previous section, but it means struct-typed variables store their data directly, while class-typed variables store a reference, and the actual data lives elsewhere. (Now might be a good time to re-read Level 14 if you're still struggling with these differences.)

This one difference has a lot of ramifications, not the least of which is the differences in constructors described in the previous section.

Another key difference is that structs cannot take on a null value, though we will see a way to pretend in Level 32.

Because structs are value types, reading and writing values to variables involves copying the whole pile of data around, not just a reference. Like with a **`double`**, when we copy a value from one variable to another results in a copy:

```
PairOfInts first = new PairOfInts(2, 10);
PairOfInts second = first;
```

Here, **`second`** will get a copy of both the **`2`** and the **`10`** assigned to its fields. The same thing would happen if we passed a **`PairOfInts`** to a method as an argument.

Additionally, inheritance does not work well when copying value types around (do a web search for "object slicing" if you want to know more), so structs do not support it. A struct cannot pick a base class (they all derive from **`ValueType`**, which derives from **`object`**). Structs, however, are allowed to implement interfaces.

Equality is also different for structs. As we saw in Level 14, value types have value semantics—two things are equal if all of their data members are equal. Any struct you create will automatically have value semantics. The **`Equals`** method and the **`==`** and **`!=`** operators are automatically defined to compare the struct's fields for equality.

Choosing to Make a Class or a Struct

Given how similar structs and classes are, you're probably wondering how to decide between the two. Ultimately, the deciding factor should be if you need a reference type or a value type. That's the main difference, and it should drive your selection.

Structs are usually the better choice for small, data-focused types. A struct may be better if a concept is primarily about representing data and not doing work. If the concept's behavior is important, then things like inheritance and polymorphism often are as well. You can't get that from a struct. That doesn't mean a struct can't have methods, but a struct's methods are usually focused on answering questions about the data instead of getting work done.

However, just because something focuses on data doesn't mean a struct is always better. You can't get references to a struct like you can with a class. With a class, you can build a web of interconnected objects that know about each other through references. You can't do the same thing with structs.

The way structs and classes are managed in memory is also a driving force. Reference types like classes always get allocated individually on the heap. Structs get allocated directly in whatever contains them. That is sometimes the stack and sometimes a larger object on the heap (such as an array or class with value-typed fields). Therefore, instances of classes make the garbage collector work harder, while structs don't.

Let's illustrate that point with an example. Let's say we have the following two types that differ only by whether they are a struct (a value type) or a class (a reference type):

```
public struct CircleStruct
{
    public double X { get; }
    public double Y { get; }
    public double Radius { get; }

    public CircleStruct(double x, double y, double radius)
    {
```

```
        X = x; Y = y; Radius = radius;
    }
}

public class CircleClass
{
    public double X { get; }
    public double Y { get; }
    public double Radius { get; }

    public CircleClass(double x, double y, double radius)
    {
        X = x; Y = y; Radius = radius;
    }
}
```

Consider this code:

```
for (int number = 0; number < 10000; number++)
{
    CircleStruct circle = new CircleStruct(0, 0, 10);
    Console.WriteLine($"X={circle.X} Y = {circle.Y} Radius={circle.Radius}");
}

for (int number = 0; number < 10000; number++)
{
    CircleClass circle = new CircleClass(0, 0, 10);
    Console.WriteLine($"X={circle.X} Y = {circle.Y} Radius={circle.Radius}");
}
```

In the first loop, with structs, there is one variable designed to hold a single **`CircleStruct`**, and because it is a local variable, it lives on the stack. That variable is big enough to contain an entire **`CircleStruct`**, with 8 bytes for **`X`**, **`Y`**, and **`Radius`** for a total of 24 bytes. Every time we get to that **`new CircleStruct(...)`** part, we re-initialize that memory location with new data. But we reuse the memory location.

In the second loop, with classes, we still have a single variable on the stack, but that variable is a reference type and will only hold references. This variable will be only 8 bytes (on a 64-bit computer). However, each time we run **`new CircleClass(...)`**, a new **`CircleClass`** object is allocated on the heap. By the time we finish, we will have done that 10,000 times (and used 240,000 bytes), and the garbage collector will need to clean them all up.

Structs don't always have the upper hand with memory usage. Consider this scenario, where we pass a circle as an argument to a method 10,000 times:

```
CircleStruct circleStruct = new CircleStruct(0, 0, 10);
for (int number = 0; number < 10000; number++)
    DisplayStruct(circleStruct);

CircleClass circleClass = new CircleClass(0, 0, 10);
for (int number = 0; number < 10000; number++)
    DisplayClass(circleClass);

void DisplayStruct(CircleStruct circle) =>
        Console.WriteLine($"X={circle.X} Y={circle.Y} Radius={circle.Radius}");

void DisplayClass(CircleClass circle) =>
        Console.WriteLine($"X={circle.X} Y={circle.Y} Radius={circle.Radius}");
```

We only create one struct and class instance here, but we repeatedly call the **DisplayStruct** and **DisplayClass** methods. In doing so, the contents of **circleStruct** are copied to **DisplayStruct**'s **circle** parameter, and the contents of **circleClass** are copied to **DisplayClass**'s **circle** parameter repeatedly. For the struct, that means copying all 24 bytes of the data structure, for a total of 240,000 bytes copied. For the class, we're only copying the 8-byte reference and a total of 80,000 bytes, which is far less.

The bottom line is that you'll get different memory usage patterns depending on which one you pick. Those differences play a key role in deciding whether to choose a class or a struct.

In short, you should consider a struct when you have a type that (1) is focused on data instead of behavior, (2) is small in size, (3) where you don't need shared references, and (4), and when being a value type works to your advantage instead of against you. If any of those are not true, you should prefer a class.

To give a few more examples, a point, rectangle, circle, and score could each potentially fit those criteria, depending on how you're using them.

I'll let you in on a secret: many C# programmers, including some veterans, don't fully grasp the differences between a class and a struct and will always make a class. I don't think this is ideal, but it may not be so bad as a short-term strategy as you get more comfortable in C#.

Just don't let that be your permanent strategy. I probably make 50 times as many classes as structs, but a few strategically placed structs make a big difference.

Rules to Follow When Making Structs

There are three guidelines that you should follow when you make a struct.

First, keep them small. That is subjective, but an 8-byte struct is fine, while a 200-byte struct should generally be avoided. The costs of copying large structs add up.

Second, make structs immutable. Structs should represent a single compound value, and as such, you should make its fields **readonly** and not have setters (not even private) for its properties. (An **init** accessor is fine.) Doing this helps prevent situations where somebody thought they had modified a struct value but modified a copy instead:

```
public void ShiftLeft(Point p) => p.X -= 10;
```

Assuming **Point** is a struct, the data is copied into **p** when you call this method. The variable **p**'s **X** property is shifted, but it is **ShiftLeft**'s copy. The original copy is unaffected.

Making structs immutable sidesteps all sorts of bugs like this. If you want to shift a point to the left, you make a new **Point** value instead, with its **X** property adjusted for the desired shift. Making a new value is essentially the same thing you would do if it were just an **int**.

```
public Point ShiftLeft(Point p) => new Point(p.X - 10, p.Y);
```

With this change, the calling method would do this:

```
Point somePoint = new Point(5, 5);
somePoint = ShiftLeft(somePoint);
```

Third, because struct values can exist without calling a constructor, a default, zeroed-out struct should represent a valid value. Consider the **LineSegment** class below:

```
public class LineSegment
{
    private readonly Point _start;
    private readonly Point _end;

    public LineSegment() { }

    // ...
}
```

When a new **LineSegment** is created, **_start** and **_end** are initialized to all zeroes. Regardless of what constructors **Point** defines, they don't get called here. Fortunately, a **Point** whose **X** and **Y** values are 0 represents a point at the origin, which is a valid point.

BUILT-IN TYPE ALIASES

The built-in types that are value types (all eight integer types, all three floating-point types, **char**, and **bool**) are not just value types but structs themselves.

While we have used keywords (**int**, **double**, **bool**, **char**, etc.) to refer to these types, the keywords are aliases or shortcut names for their formal struct names. For example, **int** is an alias for **System.Int32**. While rarely done, we could use these other names instead:

```
Int32 x = new Int32();
Int32 y = 0;              // Or combined.
int z = new Int32();      // Or combined another way. It's all the same thing.
int w = new int();        // Yet another way...
```

The keyword version is simpler and nearly always preferred, but their aliases pop up from time to time in documentation and sometimes in Visual Studio. Knowing the long name for these types can help you understand what is going on. Here is the complete list of these aliases:

Built-In Type	Alias For:	Class or Struct?
bool	**System.Boolean**	struct
byte	**System.Byte**	struct
sbyte	**System.SByte**	struct
char	**System.Char**	struct
decimal	**System.Decimal**	struct
double	**System.Double**	struct
float	**System.Single**	struct
int	**System.Int32**	struct
uint	**System.UInt32**	struct
long	**System.Int64**	struct
ulong	**System.UInt64**	struct
object	**System.Object**	class
short	**System.Int16**	struct
ushort	**System.UInt16**	struct
string	**System.String**	class

Ignoring the **System** part, many of these are the same except for capitalization. C# keywords are all lowercase, while types are usually UpperCamelCase, which explains that difference.

These names follow the same naming pattern we saw with **`Convert`**'s various methods. (**`Convert`**'s method names actually come from these names, not the other way around.)

But the keyword and the longer type name are true synonyms. The following two are identical:

```
int.Parse("4");
Int32.Parse("4");
```

BOXING AND UNBOXING

Classes and structs all ultimately share the same base class: **`object`**. Classes derive from **`object`** directly (unless they choose another base class), while structs derive from the special **`System.ValueType`** class, which is derived from **`object`**. This creates an interesting situation:

```
object thing = 3;
int number = (int)thing;
```

Some fascinating things are going on here. The number **`3`** is an **`int`** value, and **`int`**-typed variables contain the value directly, rather than a reference. But variables of the **`object`** type store references. It seems we have conflicting behaviors. How does the above code work?

When a struct value is assigned to a variable that stores references, like the first line above, the data is pushed out to another location on the heap, in its own little container—a *box*. A reference to the box is then stored in the **`thing`** variable. This is called a *boxing conversion*. The value is copied onto the heap, allowing you to grab a reference to it.

On the second line, the inverse happens. After ensuring that the type is correct, the box's contents are extracted—an *unboxing conversion*—and copied into the **`number`** variable.

You might hear a C# programmer phrase this as, "The 3 is boxed in the first line, and then unboxed on the second line."

As shown above, boxing can happen implicitly, while unboxing must be explicit with a cast.

The same thing happens when we use an interface type with a value type. Suppose a value type implements an interface, and you store it in a variable that uses an interface type. In that case, it must box the value before storing it because interface types store references.

```
ISomeInterface thing = new SomeStruct();
SomeStruct s = (SomeStruct)thing;
```

Boxing and unboxing are efficient but not free. If you are boxing and unboxing frequently, perhaps you should make it a class instead of a struct.

Challenge **Room Coordinates** **50 XP**

The time to enter the Fountain of Objects draws closer. While you don't know what to expect, you have found some scrolls that describe the area in ancient times. It seems to be structured as a set of rooms in a grid-like arrangement.

Locations of the room may be represented as a row and column, and you take it upon yourself to try to capture this concept with a new struct definition.

Objectives:

- Create a **`Coordinate`** struct that can represent a room coordinate with a row and column.

- Ensure **`Coordinate`** is immutable.
- Make a method to determine if one coordinate is adjacent to another (differing only by a single row or column).
- Write a main method that creates a few coordinates and determines if they are adjacent to each other to prove that it is working correctly.

Level 29

Records

Speedrun

- Records are a compact alternative notation for defining a data-centric class or struct: `public record Point(float X, float Y);`
- The compiler automatically generates a constructor, properties, `ToString`, equality with value semantics, and deconstruction.
- You can add additional members or provide a definition for most compiler-synthesized members.
- Records are turned into classes by default or into a struct (`public record struct Point(...)`).
- Records can be used in a `with` expression: `Point modified = p with { X = -2 };`

Records

C# has an ultra-compact way to define certain kinds of classes or structs. This compact notation is called a *record*. The typical situation where a record makes sense is when your type is little more than a set of properties—a data-focused entity.

The following shows a simple `Point` record, defined with an `X` and `Y` property:

```
public record Point(float X, float Y); // That's all.
```

The compiler will expand the above code into something like this:

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

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

When you define a record, you get several features for free. It starts with properties that match the names you provided in the record definition and a matching constructor. Note that these

properties are **`init`** properties, so the class is, by default, immutable. But that's only the beginning. We get several other things for free: a nice string representation, value semantics, deconstruction, and creating copies with tweaks. We'll look at each of these features below.

String Representation

Records automatically override the **`ToString`** method with a convenient, readable representation of its data. For example, **`new Point(2, 3).ToString()`**, will produce this:

```
Point { X = 2, Y = 3 }
```

When a type's data is the focus, a string representation like this is a nice bonus. You could do this manually by overriding **`ToString`** (Level 26), but we get it free with records.

Value Semantics

Recall that value semantics are when the thing's value or data counts, not its reference. While structs have value semantics automatically, classes have reference semantics by default. However, records automatically have value semantics. In a record, the **`Equals`** method, the **`==`** operator, and the **`!=`** operator are redefined to give it value semantics. For example:

```
Point a = new Point(2, 3);
Point b = new Point(2, 3);
Console.WriteLine(a == b);
```

Though **`a`** and **`b`** refer to different instances and use separate memory locations, this code displays **`True`** because the data are a perfect match, and the two are considered equal. Level 41 describes making operators for your own types, but we get it for free with a record.

Deconstruction

In Level 17, we saw how to deconstruct a tuple, unpacking the data into separate variables:

```
(string first, string last) = ("Jack", "Sparrow");
```

You can do the same thing with records:

```
Point p = new Point(-2, 5);
(float x, float y) = p;
```

In Level 34, we will see how you can add deconstruction to any type, but records get it for free.

`with` Statements

Given that records are immutable by default, it is not uncommon to want a second copy with most of the same data, just with one or two of the properties tweaked. While you could always just call the constructor, passing in the right values, records give you extra powers in the form of a **`with`** statement:

```
Point p1 = new Point(-2, 5);
Point p2 = p1 with { X = 0 };
```

You can replace many properties at once by separating them with commas:

```
Point p3 = p1 with { X = 0, Y = 0 };
```

In this case, since we've replaced *all* the properties with new values, it might have been better just to write **`new Point(0, 0)`**, but that code shows the mechanics.

The plumbing that the compiler generates to facilitate the **`with`** statement is not something you can add to your own types. This is a record-only feature (at least for now).

ADVANCED SCENARIOS

Most records you define will be a single line, similar to the **`Point`** record defined earlier. But when you have the need, they can be much more. You can add additional members and make your own definition to supplant most compiler-generated members.

Additional Members

In any record, you can add any members you need to flesh out your record type, just like a class. The following shows a **`Rectangle`** record with **`Width`** and **`Height`** properties and then adds in an **`Area`** property, calculated from the rectangle's width and height:

```
public record Rectangle(float Width, float Height)
{
    public float Area => Width * Height;
}
```

There are no limits to what members you can add to a record.

Replacing Synthesized Members

The compiler generates quite a few members to provide the features that make records attractive. While you can't remove any of those features, you can customize most of them to meet your needs. For example, as we saw, the **`Point`** record defines **`ToString`** to display text like **`Point { X = 2, Y = 3 }`**. If you wanted your **`Point`** record to show it like **`(2, 3)`** instead, you could simply add in your own definition for **`ToString`**:

```
public record Point(float X, float Y)
{
    public override string ToString() => $"({X}, {Y})";
}
```

In most situations where the compiler would normally synthesize a member for you, if it sees that you've provided a definition, it will use your version instead.

One use for this is defining the properties as mutable properties or fields instead of the default **`init`**-only property. The compiler will not automatically assign initial values to your version if you do this. You'll want to initialize them yourself:

```
public record Point(float X, float Y)
{
    public float X { get; set; } = X;
}
```

You cannot supply a definition for the constructor (though this limitation is removed if you make a non-positional record, as described later in this section).

You cannot define many of the equality-related members, including **`Equals(object)`**, the **`==`** operator, and the **`!=`** operator. However, you can define **`Equals(Point)`**, or whatever the

record's type is. **`Equals(object)`**, **`==`**, and **`!=`** each call **`Equals(Point)`**, so you can usually achieve what you want, despite this limitation.

Non-Positional Records

Most records will include a set of properties in parentheses after the record name. These are positional records because the properties have a known, fixed ordering (which also matters for deconstruction). These parameters are not strictly required. You could also write a simple record like this:

```
public record Point
{
    public float X { get; init; }
    public float Y { get; init; }
}
```

In this case, you wouldn't get the constructor or the ability to do deconstruction (unless you add them in yourself), but otherwise, this is the same as any other record.

STRUCT- AND CLASS-BASED RECORDS

The compiler turns records into classes by default because this is the more common scenario. However, you can also make a record struct instead:

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

This code will now generate a struct instead of a class, bringing along all the other things we know about structs vs. classes (in particular, this is a value type instead of a reference type).

A record struct creates properties slightly different from class-based structs. They are defined as **`get`**/**`set`** properties instead of **`get`**/**`init`**. The record struct above becomes something more like this:

```
public struct Point
{
    public float X { get; set; }
    public float Y { get; set; }

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

Records are class-based, by default, but if you want to call it out specifically, you can write it out explicitly:

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

This definition is no different than if it were defined without the **`class`** keyword, other than drawing a bit more attention to the choice of making the record class-based.

Whichever way you go, you can generally expect the same things of a record as you can of the class or struct it would become. For example, since you can make a class **`abstract`** or **`sealed`**, those are also options for class-based records.

Inheritance

Class-based records can also participate in inheritance with a few limitations. Records cannot derive from normal classes, and normal classes cannot derive from records.

The syntax for inheritance in a record is worth showing:

```
public record Point(float X, float Y);
public record ColoredPoint(Color Color, float X, float Y) : Point(X, Y);
```

When to Use a Record

When defining a class or a struct, you have the option to use the record syntax. So when should you make a record, and when should you create a normal class or struct?

The record syntax conveys a lot of information in a very short space. If the feature set of records fits your needs, you should generally prefer the record syntax. Records give you a concise way to make a type with several properties and a constructor to initialize them. They also give you a nice string representation, value semantics, deconstruction, and the ability to use **`with`** statements. If that suits your needs, a record is likely the right choice. If those features get in your way or are unhelpful, then a regular class or struct is the better choice.

You should also consider records as a possible alternative to tuples. I usually go with a record in these cases. You need to go to the trouble of formally defining the record type, but you get actual names for the type and its members. For me, that is usually worth the small cost.

Fortunately, it isn't usually hard to swap out one of these options for another. If you change your mind, you can change the code. (And your intuition will get better with practice.)

Challenge War Preparations 100 XP

As you pass through the city of Rocaard, two blacksmiths, Cygnus and Lyra, approach you. "We know where this is headed. A confrontation with the Uncoded One's forces," Lyra says. Cygnus continues, "You're going to need an army at your side—one prepared to do battle. We forge enchanted swords and will do everything we can to support this cause. We need the Power of Programming to flow unfettered too. We want to help, but we can't equip an entire army without the help of a program to aid in crafting swords." They describe the program they need, and you dive in to help.

Objectives:

- Swords can be made out of any of the following materials: wood, bronze, iron, steel, and the rare binarium. Create an enumeration to represent the material type.
- Gemstones can be attached to a sword, which gives them strange powers through Cygnus and Lyra's touch. Gemstone types include emerald, amber, sapphire, diamond, and the rare bitstone. Or no gemstone at all. Create an enumeration to represent a gemstone type.
- Create a **`Sword`** record with a material, gemstone, length, and crossguard width.
- In your main program, create a basic **`Sword`** instance made out of iron and with no gemstone. Then create two variations on the basic sword using **`with`** expressions.
- Display all three sword instances with code like **`Console.WriteLine(original);`**.

Level 30

Generics

Speedrun

- Generics solve the problem of making classes or methods that would differ only by the types they use. Generics leave placeholders for types that can be filled in when used.
- Defining a generic class: `public class List<T> { public T GetItemAt(int index) { ... } ... }`
- You can also make generic methods and generic types with multiple type parameters.
- Constraints allow you to limit what can be used for a generic type argument while enabling you to know more about the types being used: `class List<T> where T : ISomeInterface { }`

We'll look at a powerful feature in C# called *generics* (*generic types* and *generic methods*) in this level. We'll start with the problem this feature solves and then see how generics solve it. In Level 32, we will see a few existing generic types that will make your life a lot easier.

The Motivation for Generics

By now, you've probably noticed that arrays have a big limitation: you can't easily change their size by adding and removing items. The best you can do is copy the contents of the array to a new array, making any necessary changes in the process, and then update your array variable:

```
int[] numbers = new int[] { 1, 2, 3 };
numbers = AddToArray(numbers, 4);

int[] AddToArray(int[] input, int newNumber)
{
    int[] output = new int[input.Length + 1];

    for (int index = 0; index < input.Length; index++)
        output[index] = input[index];

    output[^1] = newNumber;

    return output;
}
```

With your understanding of objects and classes, you might say to yourself, "I could make a class that handles this for me. Then whenever I need it, I can just use the class instead of an array, and growing and shrinking the collection will happen automatically." Indeed, this would make a great reusable class. What an excellent idea! You start with this:

```
public class ListOfNumbers
{
    private int[] _numbers = new int[0];

    public int GetItemAt(int index) => _numbers[index];
    public void SetItemAt(int index, int value) => _numbers[index] = value;

    public void Add(int newValue)
    {
        int[] updated = new int[_numbers.Length + 1];

        for (int index = 0; index < _numbers.Length; index++)
            updated[index] = _numbers[index];

        updated[^1] = newValue;

        _numbers = updated;
    }
}
```

This **ListOfNumbers** class has a field that is an **int** array. It includes methods for getting and setting items at a specific index in the list. Also, it includes an **Add** method, which tacks a new **int** to the end of the collection, copying everything over to a new, slightly longer array, and placing the new value at the end. The code in **Add** is essentially the same as our **AddToArray** method earlier. I won't add code for removing an item, but you could do something similar.

Now we can use this class like this:

```
ListOfNumbers numbers = new ListOfNumbers();
numbers.Add(1);
numbers.Add(2);
numbers.Add(3);
Console.WriteLine(numbers.GetItemAt(2));
```

This is a better solution because it is object-oriented. Instead of having a loose array and a loose method to work with it, the two are combined. The class handles growing the collection as needed, and the outside world is free to assume it does the job assigned to it. And it is reusable! With this class defined, any time we want a growable collection of **int**s, we make a new instance of **ListOfNumbers**, and off we go.

I do have one complaint. With arrays, you can use the indexing operator. **numbers[0]** is cleaner than **numbers.GetItemAt(0)**. We can solve that problem with the tools we'll learn in Level 41. For now, we'll just live with it.

However, there's a second, more substantial problem. We can make instances of **ListOfNumbers** whenever we want, but what if we need it to be **string**s instead? **ListOfNumbers** is built around **int**s. It is useless if we need the **string** type.

Using only tools we already know, we have two options. We could just create a **ListOfStrings** class:

```
public class ListOfStrings
{
    private string[] _strings = new string[0];

    public string GetItemAt(int index) => _strings[index];
    public void SetItemAt(int index, string value) => _strings[index] = value;

    public void Add(string newValue) { /* Details skipped */ }
}
```

This has potential, though it isn't great. What if we need a list of **`bool`**s? A list of **`double`**s? A list of points? A list of **`int[]`**? How many of these do we make? We would have to copy and paste this code repeatedly, making tiny tweaks to change the type each time. In Level 23, we said that designs with duplicate code are worse than ones that do not. This approach results in a lot of duplicate code. Imagine making 20 of these, only to discover a bug in them!

The second approach would be just to use **`object`**. With **`object`**, we can use it for anything:

```
public class List
{
    private object[] _items = new object[0];

    public object GetItemAt(int index) => _items[index];
    public void SetItemAt(int index, object value) => _items[index] = value;

    public void Add(object newValue) { /* Details skipped */ }
}
```

Which could get used like this:

```
List numbers = new List();
numbers.Add(1);
numbers.Add(2);

List words = new List();
words.Add("Hello");
words.Add("World");
```

Unfortunately, this also has a couple of big drawbacks. The first is that the **`GetItemAt`** method (and others) return an **`object`**, not an **`int`** or a **`string`**. We must cast it:

```
int first = (int)numbers.GetItemAt(0);
```

The second drawback is that we have thrown out all type checking that the compiler would normally help us with. Consider this code, which compiles but isn't good:

```
List numbers = new List();
numbers.Add(1);
numbers.Add("Hello");
```

Do you see the problem? From its name, **`numbers`** should contain only numbers. But we just dropped a **`string`** into it. The compiler cannot detect this because we are using **`object`**, and **`string`** is an **`object`**. This code won't fail until you cast to an **`int`**, expecting it to be one, only to discover it was a **`string`**.

Neither of these solutions is perfect. But this is where generics save the day.

DEFINING A GENERIC TYPE

A generic type is a type definition (class, struct, or interface) that leaves a placeholder for some of the types it uses. This is conceptually similar to making methods with parameters, allowing the outside world to supply a value. The easiest way to show a generic type is with an example of a generic **`List`** class:

```
public class List<T>
{
    private T[] _items = new T[0];

    public T GetItemAt(int index) => _items[index];
    public void SetItemAt(int index, T value) => _items[index] = value;

    public void Add(T newValue)
    {
        T[] updated = new T[_items.Length + 1];

        for (int index = 0; index < _items.Length; index++)
            updated[index] = _items[index];

        updated[^1] = newValue;

        _items = updated;
    }
}
```

Before going further, I'm going to interrupt with an important note. The code above defines our own custom generic **`List`** class. You might be thinking, "I can use something like this!" But there is already an existing generic **`List`** class that does all of this and more, is well tested, and is optimized. This code illustrates generic types, but once we learn about the official **`List<T>`** class (Level 32), we should be using that instead. Now back to our discussion.

When defining the class, we can identify a placeholder for a type in angle brackets (that **`<T>`** thing). This placeholder type is called a *generic type parameter*. It is like a method parameter, except it works at a higher level and stands in for a specific type that will be chosen later. It can be used throughout the class, as is done in several places in the above code. When this **`List<T>`** class is used, that code will supply the specific type it needs instead of **`T`**. For example:

```
List<int> numbers = new List<int>();
numbers.Add(1);
numbers.Add(2);
```

In this case, **`int`** is used as the *generic type argument* (like passing an argument to a method when you call it). Here, **`int`** will be used in all the places that **`T`** was listed. That means the **`Add`** method will have an **`int`** parameter, and **`GetItemAt`** will return an **`int`**.

Without defining additional types, we can use a different type argument such as **`string`**:

```
List<string> words = new List<string>();
words.Add("Hello");
words.Add("World");
```

Importantly, this potential problem is now caught by the compiler:

```
words.Add(1); // ERROR!
```

The variable **words** is a **List<string>**, not a **List<int>**. The compiler can recognize the type-related issue and flag it. We have created the best of both worlds: we only need to create a single type but can still retain type safety.

By the way, many C# programmers will read or pronounce **List<T>** as "list of **T**" and **List<int>** as "list of **int**."

There was nothing magical about the name **T**. We could have called it **M**, **Type**, or **_wakawaka**. However, there are two conventions for type names: single, capital letters (**T**, **K**, **V**, etc.) or a capital **T** followed by some more descriptive word or phrase, like **TItem**. If you only have a single generic type parameter, **T** is virtually universal.

Multiple Generic Type Parameters

You can also have multiple generic type parameters by listing them in the angle brackets, separated by commas:

```
public class Pair<TFirst, TSecond>
{
    public TFirst First { get; }
    public TSecond Second { get; }

    public Pair(TFirst first, TSecond second)
    {
        First = first;
        Second =  second;
    }
}
```

Which could be used like this:

```
Pair<string, double> namedNumber = new Pair<string, double>("pi", 3.1415926535);
Console.WriteLine($"{namedNumber.First} is {namedNumber.Second}");
```

Generic types can end up with rather complicated names. **Pair<string, double>** is a long name, and it could be worse. Instead of **string**, it could be a **List<string>** or even another **Pair<int, int>**. This results in nested generic types with extremely long names: **Pair<Pair<int, int>, double>**. While I have been avoiding **var** for clarity in this book, long, complex names like this are why some people prefer **var** or **new()** (without listing the type); without it, this complicated name shows up twice, making the code hard to understand.

Inheritance and Generic Types

Generic classes and inheritance can be combined. A generic class can derive from normal non-generic classes or other generic classes, and normal classes can be derived from generic classes. When doing this, you have some options for handling generic types in the base class. The simplest thing is just to keep the generic type parameter *open*:

```
public class FancyList<T> : List<T>
{ ... }
```

The base class's generic type parameter stays as a generic type parameter in the derived class.

Or a derived class can *close* the generic type parameter, resulting in a derived class that is no longer generic:

```
public class Polygon : List<Point>
{ ... }
```

With this definition, **`Polygon`** is a subtype of **`List<Point>`**, but you cannot make polygons using anything besides **`Point`**. The generic-ness is gone.

Of course, you can close some generic types, leave others open, and simultaneously introduce additional generic type parameters. Tricky situations like these are rare, though.

GENERIC METHODS

Sometimes, it isn't a type that needs to be generic but a single method. You can define generic methods by putting generic type parameters after a method's name but before its parentheses:

```
public static List<T> Repeat<T>(T value, int times)
{
    List<T> collection = new List<T>();

    for (int index = 0; index < times; index++)
        collection.Add(value);

    return collection;
}
```

You can use generic type parameters for method parameters and return types, as shown above. You can then use this like so:

```
List<string> words = Repeat<string>("Alright", 3);
List<int> zeroes = Repeat<int>(0, 100);
```

Generic methods do not have to live in a generic type. They can, and often are, defined in regular non-generic types.

When using a generic method, the compiler can often infer the types you use based on the parameters you pass into the method itself. For example, because **`Repeat<string>("Alright", 3)`** passes in a **`string`** as the first parameter, the compiler can tell that you want to use **`string`** as your generic type argument, and you can leave it out:

```
List<string> words = Repeat("Alright", 3);
```

You usually only need to list the generic type argument when the compiler either can't infer the type or is inferring the wrong type.

GENERIC TYPE CONSTRAINTS

By default, any type can be used as an argument for a generic type parameter. The tradeoff is that within the generic type, little is known about what type will be used, and therefore, the generic type can do little with it. For our **`List<T>`** class, this was not a problem. It was just a container to hold several items of the same type. On the other hand, if we constrain or limit the possible choices, we can know more about the type being used and do things with it.

To show an example, let's go back to our *Asteroids* type hierarchy. We had several different classes that all derived from a **`GameObject`** class. Let's say **`GameObject`** had an **`ID`** property used to identify each thing in the game uniquely:

```
public abstract class GameObject
{
    public int ID { get; }
    // ...
}
```

If we give a generic type a constraint that it must be derived from **GameObject**, then we will know that it is safe to use any of the members **GameObject** defines:

```
public class IDList<T> where T : GameObject
{
    private T[] items = new T[0];

    public T? GetItemByID(int idToFind)
    {
        foreach (T item in items)
            if (item.ID == idToFind)
                return item;

        return null;
    }

    public void Add(T newValue) { /* ... */ }
}
```

That **where T : GameObject** is called a *generic type constraint*. It allows you to limit what type arguments can be used for the given type parameter. **IDList** is still a generic type. We can create an **IDList<Asteroid>** that ensures only asteroids are added or an **IDList<Ship>** that can only use ships. But we can't make an **IDList<int>** since **int** isn't derived from **GameObject**. We reduce how generic the **IDList** class is but increase what we know about things going into it, allowing us to do more with it.

If you have several type parameters, you can constrain each of them with their own **where**:

```
public class GenericType<T, U> where T : GameObject
                               where U : Asteroid
{
    // ...
}
```

There are many different constraints you can place on a generic type parameter. The above, where you list another type that the argument must derive from, is perhaps the simplest.

You can also use the **class** and **struct** constraints to demand that the argument be either a class (or a reference type) or a struct (or a value type): **where T : class**. The **class** constraint will assume usages of the generic type parameter do not allow null as an option. By comparison, the **class?** constraint will assume usages of the generic type parameter allow null as an option.

There is also a **new()** constraint (**where T : new()**), which limits you to using only types that have a parameterless constructor. This allows you to create new instances of the generic type parameter (**new T()**). Interestingly, there is no option for other constructor constraints. The parameterless constructor is the only one.

You can also define constraints in relation to other generic type parameters if you have more than one: **public class Generic<T, U> where T : U { ... }**, or even **where T : IGenericInterface<U>**. This is rare but useful in situations that need it.

Three other constraints deal with future topics. The **unmanaged** constraint demands that the thing be an unmanaged type (Level 46). The **struct?** allows for nullable structs (Level 32). The **nonnull** constraint is like a combination of **class** and **struct** constraints (without the question marks), allowing for anything that is not null.

You don't need to memorize all of these different constraints. You'll spend far more time working with generic types than making them (Level 32). When you make a generic type, most of the time, you either won't have any constraints or use a simple **where T : SomeSpecificType**. Just remember that there are many kinds of constraints, giving you control of virtually any important aspect of the types being used as a generic type argument.

Multiple Constraints

You can define multiple constraints for each generic type parameter by separating them with commas. For example, the following requires **T** to have a parameterless constructor and to be a **GameObject**:

```
public class Factory<T> where T : GameObject, new() { ... }
```

Within this **Factory<T>** class, you would be able to create new instances of **T** because of the **new()** constraint and use any properties or methods on **GameObject**, such as **ID**, because of the **GameObject** constraint. Each constraint limits what types can be used for **T** and gives you more power within the class to do useful stuff with **T** because you know more about it.

Not every constraint can be combined with every other constraint. This limitation is either because two constraints conflict or one is made redundant by another. For example, you can't use both the **class** and **struct** constraints simultaneously. Also, you can't combine the **struct** and **new()** constraints because the **struct** constraint already guarantees you have a public, parameterless constructor.

The ordering of generic type constraints also matters. For example, calling out a specific type (like **GameObject** above) is expected to come first, while **new()** must be last. The rules are hard to describe and remember; it is usually easiest to just write them out and let the compiler point out any problems. In truth, you will only rarely run into issues like this; multiple generic type constraints are rare.

Constraints on Methods

Generic type constraints can also be applied to methods by listing them after the method's parameter list but before its body:

```
public static List<T> Repeat<T>(T value, int times) where T : class { ... }
```

THE DEFAULT OPERATOR

When using generic types, you may find some uses for the **default** operator, which allows you to get the default value for any type. (This isn't just limited to generic types and methods, but it is perhaps the most useful place.)

The basic form of this operator is to place the name of the type in parentheses after it. The result will be the default value for that type. For example, **default(int)** will evaluate to **0**, **default(bool)** will evaluate to **false**, and **default(string)** will evaluate to **null**.

However, in most cases, a simple **`0`**, **`false`**, or **`null`** is simpler code that doesn't leave people scratching their heads to remember if the default for **`bool`** was **`true`** or **`false`**. If the type can be inferred, you can leave out the type and parentheses and just use a plain **`default`**.

Where **`default`** shows its power is with generics. **`default(T)`** will produce the default, regardless of what type **`T`** is. If we go back to our **`Pair<TFirst, TSecond>`**, we could make a constructor that uses default values:

```
public Pair()
{
   First = default;  // Or default(TFirst), if the compiler cannot infer it.
   Second = default; // Or default(TSecond), if the compiler cannot infer it.
}
```

This seems more useful than it is. You still know nothing about the value you just created, so you can do little with it afterward. But it does have its occasional time and place.

Challenge — Colored Items — 100 XP

You have a sword, a bow, and an axe in front of you, defined like this:

```
public class Sword { }
public class Bow { }
public class Axe { }
```

You want to associate a color with these items (or any item type). You could make **`ColoredSword`** derived from **`Sword`** that adds a **`Color`** property, but doing this for all three item types will be painstaking. Instead, you define a new generic **`ColoredItem`** class that does this for any item.

Objectives:

- Put the three class definitions above into a new project.
- Define a generic class to represent a colored item. It must have properties for the item itself (generic in type) and a **`ConsoleColor`** associated with it.
- Add a **`void Display()`** method to your colored item type that changes the console's foreground color to the item's color and displays the item in that color. (**Hint:** It is sufficient to just call **`ToString()`** on the item to get a text representation.)
- In your main method, create a new colored item containing a blue sword, a red bow, and a green axe. Display all three items to see each item displayed in its color.

Level 31

The Fountain of Objects

Speedrun

- This level contains no new C# information. It is a large multi-part program to complete to hone your programming skills.

Narrative **Arrival at the Caverns**

You have made your way to the Cavern of Objects, high atop jagged mountains. Within these caverns lies the Fountain of Objects, the one-time source of the River of Objects that gave life to this entire island. By returning the Heart of Object-Oriented Programming—the gem you received from Simula after arriving on this island—to the Fountain of Objects, you can repair and restore the fountain to its former glory.

The cavern is a grid of rooms, and no natural or human-made light works within due to unnatural darkness. You can see nothing, but you can hear and smell your way through the caverns to find the Fountain of Objects, restore it, and escape to the exit.

The cavern is full of dangers. Bottomless pits and monsters lurk in the caverns, placed here by the Uncoded One to prevent you from restoring the Fountain of Objects and the land to its former glory.

By returning the Heart of Object-Oriented Programming to the Fountain of Objects, you can save the Island of Object-Oriented Programming!

This level contains several challenges that together build the game *The Fountain of Objects*. This game is based on the classic game *Hunt the Wumpus* with some thematic tweaks.

You do not need to complete every challenge listed here. There are two ways to proceed. Option 1 is to complete the base game (described first) and then pick *two* expansions. Option 2 is to start with the solution to the main challenge I provide on the book's website and then do five expansions.`

Option 1 gives you more practice with object-oriented design and allows you to build the game in any way you see fit. Option 2 might be better for people who are still hesitant about object-oriented design, as it gives you a chance to work in somebody else's code that provides

some foundational elements as a starting point. (Though with Option #2, you will have to begin by understanding how the code works so that you can enhance it.)

I recommend reading through all of the challenges and spending a few minutes thinking of how you might solve each before deciding.

This next point cannot be understated: this is by far the most formidable challenge we have undertaken in this book and only somewhat less demanding than the *Final Battle* challenge. Completing this will take time—even if you are experienced. But the real learning comes when you get your hands dirty in the code. Expect this to take much longer than any previous challenges, but don't get hung up on it. For example, if you are genuinely stuck on some particular challenge, try the other ones instead. If you are still stuck, look at the solutions provided on the book's website to see how others solved it, then take a break for a few minutes so that you aren't copying and pasting through memorization, and give it another try. That still counts for full points.

The Main Challenge

Boss Battle **The Fountain of Objects** **500 XP**

The *Fountain of Objects* game is a 2D grid-based world full of rooms. Most rooms are empty, but a few are unique rooms. One room is the cavern entrance. Another is the fountain room, containing the Fountain of Objects.

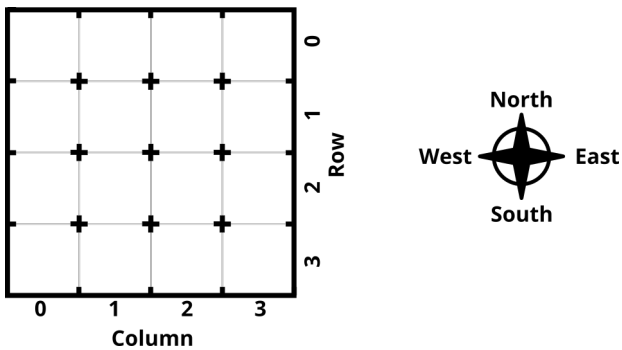
The player moves through the cavern system one room at a time to find the Fountain of Objects. They activate it and then return to the entrance room. If they do this without falling into a pit, they win the game.

Unnatural darkness pervades the caverns, preventing both natural and human-made light. The player must navigate the caverns in the dark, relying on their sense of smell and hearing to determine what room they are in and what dangers lurk in nearby rooms.

This challenge serves as the basis for the other challenges in this level. It must be completed before the others can be started. The requirements of this game are listed below.

Objectives:

- The world consists of a grid of rooms, where each room can be referenced by its row and column. North is up, east is right, south is down, and west is left:



- The game's flow works like this: The player is told what they can sense in the dark (see, hear, smell). Then the player gets a chance to perform some action by typing it in. Their chosen action is resolved

(the player moves, state of things in the game changes, checking for a win or a loss, etc.). Then the loop repeats.

- Most rooms are empty rooms, and there is nothing to sense.
- The player is in one of the rooms and can move between them by typing commands like the following: "move north", "move south", "move east", and "move west". The player should not be able to move past the end of the map.
- The room at (Row=0, Column=0) is the cavern entrance (and exit). The player should start here. The player can sense light coming from outside the cavern when in this room. ("You see light in this room coming from outside the cavern. This is the entrance.")
- The room at (Row=0, Column=2) is the fountain room, containing the Fountain of Objects itself. The Fountain can be either enabled or disabled. The player can hear the fountain but hears different things depending on if it is on or not. ("You hear water dripping in this room. The Fountain of Objects is here!" or "You hear the rushing waters from the Fountain of Objects. It has been reactivated!") The fountain is off initially. In the fountain room, the player can type "enable fountain" to enable it. If the player is not in the fountain room and runs this, there should be no effect, and the player should be told so.
- The player wins by moving to the fountain room, enabling the Fountain of Objects, and moving back to the cavern entrance. If the player is in the entrance and the fountain is on, the player wins.
- Use different colors to display the different types of text in the console window. For example, narrative items (intro, ending, etc.) may be magenta, descriptive text in white, input from the user in cyan, text describing entrance light in yellow, messages about the fountain in blue.
- An example of what the program might look like is shown below:

```
--------------------------------------------------------------------------------
You are in the room at (Row=0, Column=0).
You see light coming from the cavern entrance.
What do you want to do? move east
--------------------------------------------------------------------------------
You are in the room at (Row=0, Column=1).
What do you want to do? move east
--------------------------------------------------------------------------------
You are in the room at (Row=0, Column=2).
You hear water dripping in this room. The Fountain of Objects is here!
What do you want to do? enable fountain
--------------------------------------------------------------------------------
You are in the room at (Row=0, Column=2).
You hear the rushing waters from the Fountain of Objects. It has been reactivated!
What do you want to do? move west
--------------------------------------------------------------------------------
You are in the room at (Row=0, Column=1).
What do you want to do? move west
--------------------------------------------------------------------------------
You are in the room at (Row=0, Column=0).
The Fountain of Objects has been reactivated, and you have escaped with your life!
You win!
```

- **Hint:** You may find two-dimensional arrays (Level 12) helpful in representing a 2D grid-based game world.
- **Hint:** Remember your training! You do not need to solve this entire problem all at once, and you do not have to get it right in your first attempt. Pick an item or two to start and solve just those items. Rework until you are happy with it, then add the next item or two.

EXPANSIONS

The following six challenges extend the basic *Fountain of Objects* game in different ways. If you did the core Fountain of Objects challenge above, pick two of the following challenges. If you choose the Expansions path and start with my code from the website, complete five of the following.

Boss Battle — Small, Medium, or Large — 100 XP

The larger the Cavern of Objects is, the more difficult the game becomes. The basic game only requires a small 4×4 world, but we will add a medium 6×6 world and a large 8×8 world for this challenge.

Objectives:

- Before the game begins, ask the player whether they want to play a small, medium, or large game. Create a 4×4 world if they choose a small world, a 6×6 world if they choose a medium world, and an 8×8 world if they choose a large world.
- Pick an appropriate location for both the Fountain Room and the Entrance room.
- **Note:** When combined with the *Amaroks*, *Maelstroms*, or *Pits* challenges, you will need to adapt the game by adding amaroks, maelstroms, and pits to all three sizes.

Boss Battle — Pits — 100 XP

The Cavern of Objects is a dangerous place. Some rooms open up to bottomless pits. Entering a pit means death. The player can sense a pit is in an adjacent room because a draft of air pushes through the pits into adjacent rooms. Add pit rooms to the game. End the game if the player stumbles into one.

Objectives:

- Add a pit room to your 4×4 cavern anywhere that isn't the fountain or entrance room.
- Players can sense the draft blowing out of pits in adjacent rooms (all eight directions): "You feel a draft. There is a pit in a nearby room."
- If a player ends their turn in a room with a pit, they lose the game.
- **Note:** When combined with the *Small, Medium, or Large* challenge, add one pit to the 4×4 world, two pits to the 6×6 world, and four pits to the 8×8 world, in locations of your choice.

Boss Battle — Maelstroms — 100 XP

The Uncoded One knows the significance of the Fountain of Objects and has placed minions in the caverns to defend it. One of these is the maelstrom—a sentient, malevolent wind. Encountering a maelstrom does not result in instant death, but entering a room containing a maelstrom causes the player to be swept away to another room. The maelstrom also moves to a new location. If the player is moved to another dangerous location, such as a pit, that room's effects will happen upon landing in that room.

A player can hear the growling and groaning of a maelstrom from a neighboring room (including diagonals), which gives them a clue to be careful.

Modify the basic Fountain of Objects game in the ways below to add maelstroms to the game.

Objectives:

- Add a maelstrom to the small 4×4 game in a location of your choice.
- The player can sense maelstroms by hearing them in adjacent rooms. ("You hear the growling and groaning of a maelstrom nearby.")
- If a player enters a room with a maelstrom, the player moves one space north and two spaces east, while the maelstrom moves one space south and two spaces west. When the player is moved like this, tell them so. If this would move the player or maelstrom beyond the map's edge, ensure they stay on the map. (Clamp them to the map, wrap around to the other side, or any other strategy.)
- **Note:** When combined with the *Small, Medium, or Large* challenge, place one maelstrom into the medium-sized game and two into the large-sized game.

Boss Battle — Amaroks — 100 XP

The Uncoded One has also placed amaroks in the caverns to protect the fountain from people like you. Amaroks are giant, rotting, wolf-like creatures that stalk the caverns. When players enter a room with an amarok, they are instantly killed, and the game is over. Players can smell an amarok in any adjacent room (all eight directions), which tells them that an amarok is nearby.

Modify the basic *Fountain of Objects* game as described below.

Objectives:

- Amarok locations are up to you. Pick a room to place an amarok aside from the entrance or fountain room in the small 4×4 world.
- When a player is in one of the eight spaces adjacent to an amarok, a message should be displayed when sensing surroundings that indicate that the player can smell the amarok nearby. For example, "You can smell the rotten stench of an amarok in a nearby room."
- When a player enters a room with an amarok, the player dies and loses the game.
- **Note:** When combined with the *Small, Medium, or Large* challenge, place two amaroks in the medium level and three in the large level in locations of your choosing.

Boss Battle — Getting Armed — 100 XP

Note: Requires doing the *Maelstroms* or *Amaroks* challenge first.

The player brings a bow and several arrows with them into the Caverns. The player can shoot arrows into the rooms around them, and if they hit a monster, they kill it, and it should no longer impact the game.

Objectives:

- Add the following commands that allow a player to shoot in any of the four directions: *shoot north*, *shoot east*, *shoot south*, and *shoot west*. When the player shoots in one of the four directions, an arrow is fired into the room in that direction. If a monster is in that room, it is killed and should not affect the game anymore. They can no longer sense it, and it should not affect the player.
- The player only has five arrows and cannot shoot when they are out of arrows. Display the number of arrows the player has when displaying the game's status before asking for their action.

Boss Battle — Getting Help — 100 XP

The player should not be left guessing about how to play the game. This challenge requires adding two key elements that make playing the Fountain of Objects easier: introductory text that explains the game and a **`help`** command that lists all available commands and what they each do.

Objectives:

- When the game starts, display text that describes the game shown below:

```
You enter the Cavern of Objects, a maze of rooms filled with dangerous pits in search
of the Fountain of Objects.
Light is visible only in the entrance, and no other light is seen anywhere in the caverns.
You must navigate the Caverns with your other senses.
Find the Fountain of Objects, activate it, and return to the entrance.
```

- If you chose to do the *Pits* challenge, add the following to the description: "Look out for pits. You will feel a breeze if a pit is in an adjacent room. If you enter a room with a pit, you will die."
- If you chose to do the *Maelstroms* challenge, add the following to the description: "Maelstroms are violent forces of sentient wind. Entering a room with one could transport you to any other location in the caverns. You will be able to hear their growling and groaning in nearby rooms."
- If you chose to do the *Amaroks* challenge, add the following to the description: "Amaroks roam the caverns. Encountering one is certain death, but you can smell their rotten stench in nearby rooms."
- If you chose to do the *Getting Armed* challenge, add the following to the description: "You carry with you a bow and a quiver of arrows. You can use them to shoot monsters in the caverns but be warned: you have a limited supply."
- When the player types the command **`help`**, display all available commands and a short description of what each does. The complete list of commands will depend on what challenges you complete.

Narrative — The Fountain Remade

You scramble through the dark Cavern of Objects, crawling and feeling your way to the Fountain of Objects. The dripping sound that you hear is a giveaway that you have found it. You pull Simula's green gem—the Heart of Object-Oriented Programming—out of your pack and hold it in the palm of your hand, contemplating the journey you have taken to get here. You slide your hand along the Fountain until you find a small recess, just big enough for the Heart to be placed. You slide the green gem in, and the fountain immediately comes to life. The water in the fountain, previously still, suddenly begins churning and overflowing onto the ground around you. You make a hasty escape to the cavern entrance.

Within minutes, water rushes out the entrance and through a thousand other holes in the mountainside, collecting into a raging waterfall down into the valley below. Within days, the newly restored River of Objects will flow to the sea, restoring its life-giving power to the entire island.

With the River of Objects flowing again, the land will become bountiful with objects of every class, interface, and struct imaginable. The island has been saved. You turn your attention towards the scattered islands on the horizon and your final destination beyond: a confrontation with The Uncoded One.

Level 32

Some Useful Types

Speedrun

- **`Random`** generates pseudo-random numbers.
- **`DateTime`** gets the current time and stores time and date values.
- **`TimeSpan`** represents a length of time.
- **`Guid`** is used to store a globally unique identifier.
- **`List<T>`** is a popular and versatile generic collection—use it instead of arrays for most things.
- **`IEnumerable<T>`** is an interface for almost any collection type. The basis of **`foreach`** loops.
- **`Dictionary<TKey, TValue>`** can look up one piece of information from another.
- **`Nullable<T>`** is a struct that can express the concept of a missing value for value types.
- **`ValueTuple`** is the secret sauce behind tuples in C#.
- **`StringBuilder`** is a less memory-intensive way to build strings a little at a time.

Narrative — The Harvest of Objects

A few days have passed since the Fountain of Objects was restored, but the land has already become more vibrant and lush. New objects and classes, unseen for thousands of clock cycles, have been found again. The classes described in this level represent a collection of some of the most versatile and interesting ones you have seen, and you gather some up for the rest of your journey.

Now that we have learned about classes, structs, interfaces, and generic types, we are well prepared to look at a handful of useful types that come with .NET. There are thousands of types in C#'s standard library called the *Base Class Library* (*BCL*). We can't reasonably cover them all. We have covered several in the past and will cover more in the future, but in this level, we will look at nine types that will forever change how you program in C#.

THE RANDOM CLASS

The **Random** class (in the **System** namespace) generates random numbers. Some programs (like games) are more likely to use random numbers than others, but randomness can be found anywhere.

Randomness is an interesting concept. A computer follows instructions exactly, which does not leave room for true randomness, short of buying hardware that measures some natural random phenomenon (like thermal noise or the photoelectric effect). However, some algorithms will produce a sequence of numbers that *feels* random, based on past numbers. This is called *pseudo-random number generation* because it is not truly random. For most practical purposes, including most games, pseudo-random number generation is sufficient.

Pseudo-random generators have to start with an initial value called a *seed*. If you reuse the same seed, you will get the same random-looking sequence again precisely. This can be both bad and good. For example, *Minecraft* generates worlds based on a seed. Sometimes, you want a *specific* random world, and by telling *Minecraft* to use a particular seed, you can see the same world again. But most of the time, you want a random seed to get a unique world.

The **System.Random** class is the starting point for anything involving randomness. It is a simple class that is easy to learn how to use:

```
Random random = new Random();
Console.WriteLine(random.Next());
```

The **Random()** constructor is initialized with an arbitrary seed value, which means you will not see the same sequence come up ever again with another **Random** object or by rerunning the program. (Older versions of .NET used the current time as a seed, which meant creating two **Random** instances in quick succession would have the same seed and generate the same sequence. That is no longer true.)

Random's most basic method is the **Next()** method. **Next** picks a random non-negative (0 or positive) **int** with equal chances of each. You are just as likely to get 7 as 1,844,349,103. Such a large range is rarely useful, so a couple of overloads of **Next** give you more control. **Next(int)** lets you pick the ceiling:

```
Console.WriteLine(random.Next(6));
```

random.Next(6) will give you **0**, **1**, **2**, **3**, **4**, or **5** (but not **6**) as possible choices, with equal chances of each. It is common to add 1 to this result so that the range is 1 through 6 instead of 0 through 5. For example:

```
Console.WriteLine($"Rolling a six-sided die: {random.Next(6) + 1}");
```

The third overload of **Next** allows you to name the minimum value as well:

```
Console.WriteLine(random.Next(18, 22));
```

This will randomly pick from the values **18**, **19**, **20**, and **21** (but not **22**).

If you want floating-point values instead of integers, you can use **NextDouble()**:

```
Console.WriteLine(random.NextDouble());
```

This will give you a **double** in the range of **0.0** to **1.0**. (Strictly speaking, **1.0** won't ever come up, but **0.9999999** can.) You can stretch this out over a larger range with some simple arithmetic. The following will produce random numbers in the range 0 to 10:

```
Console.WriteLine(random.NextDouble() * 10);
```

And this will produce random numbers in the range -10 to +10:

```
Console.WriteLine(random.NextDouble() * 20 - 10);
```

The **Random** class also has a constructor that lets you pass in a specific seed:

```
Random random = new Random(3445);
Console.WriteLine(random.Next());
```

This code will always display the same output because the seed is always 3445, which lets you recreate a random sequence of numbers.

Challenge — The Robot Pilot — 50 XP

When we first made the *Hunting the Manticore* game in Level 14, we required two human players: one to set up the *Manticore*'s range from the city and the other to destroy it. With **Random**, we can turn this into a single-player game by randomly picking the range for the *Manticore*.

Objectives:

- Modify your *Hunting the Manticore* game to be a single-player game by having the computer pick a random range between 0 and 100.
- **Answer this question:** How might you use inheritance, polymorphism, or interfaces to allow the game to be either a single player (the computer randomly chooses the starting location and direction) or two players (the second human determines the starting location and direction)?

THE DATETIME STRUCT

The **DateTime** struct (in the **System** namespace) stores moments in time and allows you to get the current time. One way to create a **DateTime** value is with its constructors:

```
DateTime time1 = new DateTime(2022, 12, 31);
DateTime time2 = new DateTime(2022, 12, 31, 23, 59, 55);
```

This creates a time at the start of 31 December 2022 and at 11:59:55 PM on 31 December 2022, respectively. There are 12 total constructors for **DateTime**, each requiring different information.

Perhaps even more useful are the static **DateTime.Now** and **DateTime.UtcNow** properties:

```
DateTime nowLocal = DateTime.Now;
DateTime nowUtc   = DateTime.UtcNow;
```

DateTime.Now is in your local time zone, as determined by your computer. **DateTime.UtcNow** gives you the current time in Coordinated Universal Time or UTC, which is essentially a worldwide time, not specific to time zones, daylight saving time, etc.

A **DateTime** value has various properties to see the year, month, day, hour, minute, second, and millisecond, among other things. The following illustrates some simple uses:

```
DateTime time = DateTime.Now;
if (time.Month == 10) Console.WriteLine("Happy Halloween!");
else if (time.Month == 4 && time.Day == 1) Console.WriteLine("April Fools!");
```

There are also methods for getting new **DateTime** values relative to another. For example:

```
DateTime tomorrow = DateTime.Now.AddDays(1);
```

The **DateTime** struct is very smart, handling many easy-to-forget corner cases, such as leap years and day-of-the-week calculations. When dealing with dates and times, this is your go-to struct to represent them and get the current date and time.

THE TIMESPAN STRUCT

The **TimeSpan** struct (**System** namespace) represents a span of time. You can create values of the **TimeSpan** struct in one of two ways. Several constructors let you dictate the length of time:

```
TimeSpan timeSpan1 = new TimeSpan(1, 30, 0); // 1 hour, 30 minutes, 0 seconds.
TimeSpan timeSpan2 = new TimeSpan(2, 12, 0, 0); // 2 days, 12 hours.
TimeSpan timeSpan3 = new TimeSpan(0, 0, 0, 0, 500); // 500 milliseconds.
TimeSpan timeSpan4 = new TimeSpan(10); // 10 "ticks" == 1 microsecond
```

After reading the comments, most of these are straightforward, but the last one is notable. Internally, a **TimeSpan** keeps track of times in a unit called a *tick*, which is 0.1 microseconds or 100 nanoseconds. This is as fine-grained as a **TimeSpan** can get, but you rarely need more.

The other way to create **TimeSpan**s is with one of the various **FromX** methods:

```
TimeSpan aLittleWhile = TimeSpan.FromSeconds(3.5);
TimeSpan quiteAWhile = TimeSpan.FromHours(1.21);
```

The whole collection includes **FromTicks**, **FromMilliseconds**, **FromSeconds**, **FromHours**, and **FromDays**.

TimeSpan has two sets of properties that are worth mentioning. First is this set: **Days**, **Hours**, **Minutes**, **Seconds**, **Milliseconds**. These represent the various components of the **TimeSpan**. For example:

```
TimeSpan timeLeft = new TimeSpan(1, 30, 0);
Console.WriteLine($"{timeLeft.Days}d {timeLeft.Hours}h {timeLeft.Minutes}m");
```

timeLeft.Minutes does not return 90, since 60 of those come from a full hour, represented by the **Hours** property.

Another set of properties capture the entire timespan in the unit requested: **TotalDays**, **TotalHours**, **TotalMinutes**, **TotalSeconds**, and **TotalMillseconds**.

```
TimeSpan timeRemaining = new TimeSpan(1, 30, 0);
Console.WriteLine(timeRemaining.TotalHours);
Console.WriteLine(timeRemaining.TotalMinutes);
```

This will display:

```
1.5
90
```

Both **DateTime** and **TimeSpan** have defined several operators (Level 41) for things like comparison (**>**, **<**, **>=**, **<=**), addition, and subtraction. Plus, the two structs play nice together:

```
DateTime eventTime = new DateTime(2022, 12, 4, 5, 29, 0); // 4 Dec 2022 at 5:29am
TimeSpan timeLeft = eventTime - DateTime.Now;
```

```
// 'TimeSpan.Zero' is no time at all.
if (timeLeft > TimeSpan.Zero)
    Console.WriteLine($"{timeLeft.Days}d {timeLeft.Hours}h {timeLeft.Minutes}m");
else
    Console.WriteLine("This event has passed.");
```

The second line shows that subtracting one **DateTime** from another results in a **TimeSpan** that is the amount of time between the two. The **if** statement shows a comparison against the special **TimeSpan.Zero** value.

Challenge — Time in the Cavern — 50 XP

With **DateTime** and **TimeSpan**, you can track how much time a player spends in the Cavern of Objects to beat the game. With these tools, modify your Fountain of Objects game to display how much time a player spent exploring the caverns.

Objectives:

- When a new game begins, capture the current time using **DateTime**.
- When a game finishes (win or loss), capture the current time.
- Use **TimeSpan** to compute how much time elapsed and display that to the player.

THE GUID STRUCT

The **Guid** struct (**System** namespace) represents a globally unique identifier or GUID. (The word GUID is usually pronounced as though it rhymes with "squid.") You may find value in giving objects or items a unique identifier to track them independently from other similar objects in certain programs. This is especially true if you send information across a network, where you can't just use a simple reference. While you could use an **int** or **long** as unique numbers for these objects, it can be tough to ensure that each item has a truly unique number. This is especially true if different computers have to create the unique number. This is where the **Guid** struct comes in handy.

The idea is that if you have enough possible choices, two people picking at random won't pick the same thing. If all of humanity had a beach party and each of us went and picked a grain of sand on the beach, the chance that any of us would pick the same grain is vanishingly small. The generation of new identifiers with the **Guid** struct is similar.

To generate a new arbitrary identifier, you use the static **Guid.NewGuid()** method:

```
Guid id = Guid.NewGuid();
```

Each **Guid** value is 16 bytes (4 times as many as an **int**), ensuring plenty of available choices. But **NewGuid()** is smarter than just picking a random number. It has smarts built in that ensure that other computers won't pick the same value and that multiple calls to **NewGuid()** won't ever give you the same number again, maximizing the chance of uniqueness.

A **Guid** is just a collection of 16 bytes, but it is usually written in hexadecimal with dashes breaking it into smaller chunks like this: **10A24EC2-3008-4678-AD86-FCCCDA8CE868**. Once you know about GUIDs, you will see them pop up all over the place.

If you already have a GUID and do not want to generate a new one, there are other constructors that you can use to build a new **`Guid`** value that represents it. For example:

```
Guid id = new Guid("10A24EC2-3008-4678-AD86-FCCCDA8CE868");
```

Just be careful about inadvertently reusing a GUID in situations that could cause conflicts. Copying and pasting GUIDs can lead to accidental reuse. Visual Studio has a tool to generate a random GUID under **Tools** > **Create GUID**, and you can find similar things online.

THE LIST<T> CLASS

The **`List<T>`** class (**`System.Collections.Generic`** namespace) is perhaps the most versatile generic class in .NET. **`List<T>`** is a collection class where order matters, you can access items by their index, and where items can be added and removed easily. They are like an array, but their ability to grow and shrink makes them superior in virtually all circumstances. In fact, after this section, you should only rarely use arrays.

The **`List<T>`** class is a complex class with many capabilities. We won't look at all of them, but let's look at the most important ones.

Creating List Instances

There are many ways to create a new list, but the most common is to make an empty list:

```
List<int> numbers = new List<int>();
```

This makes a new **`List<int>`** instance with nothing in it. You will do this most of the time.

If a list has a known set of initial items, you can also use collection initializer syntax as we did with arrays:

```
List<int> numbers = new List<int>() { 1, 2, 3 };
```

This calls the same empty constructor before adding the items in the collection one at a time but is an elegant way to initialize a new list with specific items. Like we saw with object initializer syntax, where we set properties on a new object, if the constructor needs no parameters, you can also leave the parentheses off:

```
List<int> numbers = new List<int> { 1, 2, 3 };
```

Some people like the conciseness of that version; others find it strange. They both work.

Indexing

Lists support indexing, just like arrays:

```
List<string> words = new List<string>() { "apple", "banana", "corn", "durian" };
Console.WriteLine(words[2]);
```

Lists also use 0-based indexing. Accessing index 2 gives you the string **`"corn"`**.

You can replace an item in a list by assigning a new value to that index, just like an array:

```
words[0] = "avocado";
```

When we made our own **`List<T>`** class in Level 30, we didn't get this simple indexing syntax, though that was because we just didn't know the right tools yet (Level 41).

Adding and Removing Items from List

A key benefit of lists over arrays is the easy ability to add and remove items. For example:

```
List<string> words = new List<string>();
words.Add("apple");
```

Add puts items at the back of the list. To put something in the middle, you use **Insert**, which requires an index and the item:

```
List<string> words = new List<string>() { "apple", "banana", "durian" };
words.Insert(2, "corn");
```

If you need to add or insert many items, there is **AddRange** and **InsertRange**:

```
List<string> words = new List<string>();
words.AddRange(new string[] { "apple", "durian" });
words.InsertRange(1, new string[] { "banana", "corn" });
```

These allow you to supply a collection of items to add to the back of the list (**AddRange**) or insert in the middle (**InsertRange**). I used arrays to hold those collections above, though the specific type involved is the **IEnumerable<T>** interface, which we will discuss next. Virtually any collection type implements that interface, so you have a lot of flexibility.

To remove items from the list, you can name the item to remove with the **Remove** method:

```
List<string> words = new List<string>() { "apple", "banana", "corn", "durian" };
words.Remove("banana");
```

If an item is in the collection more than once, only the first occurrence is removed. **Remove** returns a **bool** that tells you whether anything was removed. If you need to remove all occurrences, you could loop until that starts returning **false**.

If you want to remove the item at a specific index, use **RemoveAt**:

```
words.RemoveAt(0);
```

The **Clear** method removes everything in the list:

```
words.Clear();
```

Since we're talking about adding and removing items from a list, you might be wondering how to determine how many things are in the list. Unlike an array, which has a **Length** property, a list has a **Count** property:

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

foreach Loops

You can use a **foreach** loop with a **List<T>** as you might with an array.

```
foreach (Ship ship in ships)
    ship.Update();
```

But there's a crucial catch: you cannot add or remove items in a **List<T>** while a **foreach** is in progress. This doesn't cause problems very often, but every so often, it is painful. For example, you have a **List<Ship>** for a game, and you use **foreach** to iterate through each and let them update. While updating, some ships may be destroyed and removed. By

removing something from the list, the iteration mechanism used with **`foreach`** cannot keep track of what it has seen, and it will crash. (Specifically, it throws an **`InvalidOperationException`**; exceptions are covered in Level 35.)

There are two workarounds for this. One is to use a plain **`for`** loop. Using a **`for`** loop and retrieving the item at the current index lets you sidestep the iteration mechanism that a **`foreach`** loop uses.

```
for (int index = 0; index < ships.Count; index++)
{
    Ship ship = ships[index];
    ship.Update();
}
```

If you add or remove items farther down the list (at an index beyond the current one), there are not generally complications to adding and removing items as you go. But if you add or remove an item before the spot you are currently at, you will have to account for it. If you are looking at the item at index 3 and insert at index 0 (the start), then what was once index 3 is now index 4. If you remove the item at index 0, then what was once at index 3 is now index 2. You can use **`++`** and **`--`** to account for this, but it is a tricky situation to avoid if possible.

```
for (int index = 0; index < ships.Count; index++)
{
    Ship ship = ships[index];
    ship.Update();
    if (ship.IsDead)
    {
        ships.Remove(ship);
        index--;
    }
}
```

Another workaround is to hold off on the actual addition or removal during the **`foreach`** loop. Instead, remember which things should be added or removed by placing them in helper lists like **`toBeAdded`** and **`toBeRemoved`**. After the **`foreach`** loop, go through the items in those two helper lists and use **`List<T>`**'s **`Add`** and **`Remove`** methods to do the actual adding and removing.

Other Useful Things

The **`Contains`** method tells you if the list contains a specific item, returning **`true`** if it is there and **`false`** if not.

```
bool hasApples = words.Contains("apple");
if (hasApples)
    Console.WriteLine("Apples are already on the shopping list!");
```

The **`IndexOf`** method tells you where in a list an item can be found, or **`-1`** if it is not there:

```
int index = words.IndexOf("apple");
```

The **`List<T>`** class has quite a bit more than we have discussed here, though we have covered the highlights. At some point, you will want to use Visual Studio's AutoComplete feature or look it up on **docs.microsoft.com** and see what else it is capable of.

Challenge **Lists of Commands** **75 XP**

In Level 27, we encountered a robot with an array to hold commands to run. But we could make the robot have as many commands as we want by turning the array into a list. Revisit that challenge to make the robot use a list instead of an array, and add commands to run until the user says to stop.

Objectives:

- Change the **`Robot`** class to use a **`List<IRobotCommand>`** instead of an array for its **`Commands`** property.
- Instead of looping three times, go until the user types **`stop`**. Then run all of the commands created.

THE IENUMERABLE<T> INTERFACE

While **`List<T>`** might be the most versatile generic type, **`IEnumerable<T>`** might be the most foundational. This simple interface essentially defines what counts as a collection in .NET.

`IEnumerable<T>` defines a mechanism that allows you to inspect items one at a time. This mechanism is the basis for a **`foreach`** loop. If a type implements **`IEnumerable<T>`**, you can use it in a **`foreach`** loop.

`IEnumerable<T>` is anything that can provide an "enumerator," and the definition looks something like this:

```
public interface IEnumerable<T>
{
    IEnumerator<T> GetEnumerator();
}
```

But what's an enumerator? It is a thing that lets you look at items in a set, one at a time, with the ability to start over. It is defined roughly like this:

```
public interface IEnumerator<T>
{
    T Current { get; }
    bool MoveNext();
    void Reset();
}
```

The **`Current`** property lets you see the current item. The **`MoveNext`** method advances to the next item and returns whether there even *is* another item. **`Reset`** starts over from the beginning. Almost nobody uses an **`IEnumerator<T>`** directly. They let the **`foreach`** loop deal with it. Consider this code:

```
List<string> words = new List<string> { "apple", "banana", "corn", "durian" };

foreach(string word in words)
    Console.WriteLine(word);
```

That is equivalent to this:

```
List<string> words = new List<string> { "apple", "banana", "corn", "durian" };

IEnumerator<string> iterator = words.GetEnumerator();
```

```
while (iterator.MoveNext())
{
    string word = iterator.Current;
    Console.WriteLine(word);
}
```

List<T> and arrays both implement **IEnumerable<T>**, but dozens of other collection types also implement this interface. It is the basis for all collection types. You will see **IEnumerable<T>** everywhere.

THE DICTIONARY<TKEY, TVALUE> CLASS

Sometimes, you want to look up one object or piece of information using another. A *dictionary* (also called an *associative array* or a *map* in other programming languages) is a data type that makes this possible. A dictionary provides this functionality. You add new items to the dictionary by supplying a *key* to store the item under, and when you want to retrieve it, you provide the key again to get the item back out. The value stored and retrieved via the key is called the *value*.

The origin of the name—and an illustrative example—is a standard English dictionary. Dictionaries store words and their definitions. For any word, you can look up its definition in the dictionary. If we wanted to make an English language dictionary in C# code, we could use the generic **Dictionary<TKey, TValue>** class:

```
Dictionary<string, string> dictionary = new Dictionary<string, string>();
```

This type has two generic type parameters, one for the key type and one for the value type. Here, we used **string** for both.

We can add items to the dictionary using the indexing operator with the key instead of an **int**:

```
dictionary["battleship"] = "a large warship with big guns";
dictionary["cruiser"] = "a fast but large warship";
dictionary["submarine"] = "a ship capable of moving under the water's surface";
```

To retrieve a value, you can also use the indexing operator:

```
Console.WriteLine(dictionary["battleship"]);
```

This will display the string **"a large warship with big guns"**.

If you reuse a key, the new value replaces the first:

```
dictionary["carrier"] = "a ship that carries stuff";
dictionary["carrier"] = "a ship that serves as a floating runway for aircraft";
Console.WriteLine(dictionary["carrier"]);
```

This displays the second, longer definition; the first is gone.

What if you try to retrieve the item with a key that isn't in the dictionary?

```
Console.WriteLine(dictionary["gunship"]);
```

This blows up. (Specifically, it throws a **KeyNotFoundException**, a topic we will learn in Level 35.) We can get around this by asking if a dictionary contains a key before retrieving it:

```
if (dictionary.ContainsKey("gunship"))
    Console.WriteLine(dictionary["gunship"]);
```

Or we could ask it to use a fallback value with the **`GetValueOrDefault`** method:

```
Console.WriteLine(dictionary.GetValueOrDefault("gunship", "unknown"));
```

If you want to remove a key and its value from the dictionary, you can use the **`Remove`** method:

```
dictionary.Remove("battleship");
```

This returns a **`bool`** that indicates if anything was removed.

Once again, there is more to **`Dictionary<TKey, TValue>`** than we can cover here, though we have covered the most essential parts.

Types Besides `string`

Dictionaries are generic types, so they can use anything you want for key and value types. Strings are not uncommon, but they are certainly not the only or even primary usage.

For example, we might create a **`WordDefinition`** class that contains the definition, an example sentence, and the part of speech, and then use that in a dictionary:

```
var dictionary = new Dictionary<string, WordDefinition>();
```

The key here is still a **`string`**, while the values are **`WordDefinition`** instances. So you still look up items with **`dictionary["battleship"]`** but get a **`WordDefinition`** instance out.

Or perhaps we have a collection of **`GameObject`** instances (maybe this is the base class of all the objects in a game we're making), and each instance has an **`ID`** that is an **`int`**. We could store these in a dictionary as well, allowing us to look up the game objects by their ID:

```
Dictionary<int, GameObject> gameObjects = new Dictionary<int, GameObject>();
```

If **`GameObject`** has an **`ID`** property, you could add an item to the dictionary like this:

```
gameObjects[ship.ID] = ship;
```

This code is a good illustration of the power of generic types. We have lots of flexibility with dictionaries, which stems from our ability to pick any key or value type.

Dictionary Keys Should Not Change

Dictionaries use the *hash code* of the key to store and locate the object in memory. A hash code is a special value determined by each object, as returned by **`GetHashCode()`**, defined by **`object`**. You can override this, but for a reference type, this is based on the reference itself. For value types, it is determined by combining the hash code of the fields that compose it. Once a key has been placed in a dictionary, you should do nothing to cause its hash code to change to a different hash code. That would make it so the dictionary cannot recover the key, and the key and the object are lost for all practical purposes.

If a key is immutable, it guarantees that you won't have any problems. Types like **`int`**, **`char`**, **`long`**, and even **`string`** are all immutable, so they are safe. If a reference type, like a class, uses the default behavior, you should also be safe. But if somebody has overridden

GetHashCode, which is often done if you redefined **Equals**, **==**, and **!=**, take care not to change the key object in ways that would alter its hash code.

THE NULLABLE<T> STRUCT

The **Nullable<T>** struct (**System** namespace) lets you pretend that a value type can take on a null value. It does this by attaching a **bool HasValue** property to the original value. This property indicates whether the value should be considered legitimate. One way to work with **Nullable<T>** is like so:

```
Nullable<int> maybeNumber = new Nullable<int>(3);
Nullable<int> another = new Nullable<int>();
```

The first creates a **Nullable<int>** where the value is considered legitimate and whose value is 3, while the second is a **Nullable<int>** where the value is missing.

Nullable<T> does not create true null references. It must use value types and is a value type itself. The bytes are still allocated (plus an extra byte for the Boolean **HasValue** property). It is just that the current content isn't considered valid.

For any nullable struct, you can use its **HasValue** and **Value** properties to check if the value is legitimate or is to be considered missing, and if it is legitimate, to retrieve the actual value:

```
if (maybeNumber.HasValue)
    Console.WriteLine($"The number is {maybeNumber.Value}.");
else
    Console.WriteLine("The number is missing.");
```

But C# provides syntax to make working with **Nullable<T>** easy. You can use **int?** instead of **Nullable<int>**. You can also automatically convert from the underlying type to the nullable type (for example, to convert a plain **int** to a **Nullable<int>**) and even convert from the literal **null**. Thus, most C# programmers will use the following instead:

```
int? maybeNumber = 3;
int? another = null;
```

Nullable<T> is a convenient way to represent values when the value may be missing. But remember, this is different from null references.

Interestingly, operators on the underlying type work on the nullable counterparts:

```
maybeNumber += 2;
```

Unfortunately, that only applies to operators, not methods or properties. If you want to invoke a method or property on a nullable value, you must call the **Value** property to get a copy of the value first.

VALUETUPLE STRUCTS

We have seen many examples where the C# language makes it easy to work with some common type. As we just saw, **int?** is the same as **Nullable<int>**, and even **int** itself is simply the **Int32** struct. Tuples also have this treatment and are a shorthand way to use the **ValueTuple** generic structs. We saw how to do the following in Level 17:

```
(string, int, int) score = ("R2-D2", 12420, 15);
```

That is a shorthand version of this:

```
ValueTuple<string, int, int> score =
                new ValueTuple<string, int, int>("R2-D2", 12420, 15);
```

Most C# programmers prefer the first, simpler syntax, but sometimes the name **ValueTuple** leaks out, and it is worth knowing the two are the same thing when it does.

THE STRINGBUILDER CLASS

One problem with doing lots of operations with strings is that it has to duplicate all of the string contents in memory for every modification. Consider this code:

```
string text = "";
while (true)
{
    string? input = Console.ReadLine();
    if (input == null || input == "") break;
    text += input;
    text += ' ';
}
Console.WriteLine(text);
```

In this code, we keep creating new strings that are longer and longer. The user enters **"abc"**, and this code creates a string containing **"abc"**. It then immediately makes another string with the text **"abc "**. Then the user enters **"def"**, and your program will make another **string** containing **"abc def"** and then another containing **"abc def "**. These partial strings could get long, take up a lot of memory, and make the garbage collector work hard.

An alternative is the **StringBuilder** class in the **System.Text** namespace. **System.Text** is not one of the namespaces we get automatic access to, so the code below includes the **System.Text** namespace when referencing **StringBuilder**. (We'll address that in more depth in Level 33.) This class hangs on to fragments of strings and does not assemble them into the final string until it is done. It will get a reference to the string **"abc"** and **"def"**, but won't make any temporary combined strings until you ask for it with the **ToString()** method:

```
System.Text.StringBuilder text = new System.Text.StringBuilder();
while (true)
{
    string? input = Console.ReadLine();
    if (input == null || input == "") break;
    text.Append(input);
    text.Append(' ');
}
Console.WriteLine(text.ToString());
```

StringBuilder is an optimization to use when necessary, not something to do all the time. A few extra relatively short strings won't hurt anything. But if you are doing anything intensive, **StringBuilder** may be an easy substitute to help keep memory usage in check.