Object Oriented Programming – Classes

Disclaimer

As this is an introductory explanation of classes, I've simplified things in a few places. As you become more familiar with the concepts you may discover that things don't work 100% the way I've described below. However I've tried to stay about 99% accurate and the 1% I miss out shouldn't affect the beginner.

Our Approach

I've long felt that when the question arises, "why did they do that?", it's because there was a problem and this is the solution someone came up with. Some of the problems below are:

- Why were structs invented?
- What was wrong with structs such that classes had to be invented?
- Why do we need public and private fields in classes?
- What's wrong with public fields?

And so on.

Rather than simply stating that such-and-such works *this* way, I've tried to provide some of the motivating principles behind the designs.

Classes Are Structs (almost)

Although the Fortran language has evolved over time, for many years there was no way to group related fields[1]. For example, if your program wanted to model animals, you'd have to have separate statements that defined, say, the animal's species, its weight, number of legs, and so on. And these fields could be defined helter-skelter all over the place in your source code.

Whereas in C, you could group them into a struct. For example,

```
struct Animal {
        char    *Species;
        float   Weight;
        int     NumberOfLegs;
};
```

You could then write

```
Animal Tibbles;                 // Allocates space in memory for Tibbles
Tibbles. Species = "Cat";
Tibbles.NumberOfLegs = 4;
Tibbles.Weight = 4.7;

Animal Rover;                   // Allocate space in memory for Rover
```

[1] These days, Fortran has the STRUCTURE keyword. Finally!

```
Rover. Species = “Dog”;
Rover.NumberOfLegs = 4;
Rover.Weight = 23.9;
```

Yes, you could make do without the grouping, but it has obvious advantages (e.g. in comprehensibility) in doing it this way.

C# also has structs that work almost exactly the same way C does. But it has a generalization of the concept of a struct that it calls a class. For example, we could write

```
class Animal {
        public string Species;     // See note on “public” below
        public float Weight;
        public int NumberOfLegs;
}
Animal Cat = new Animal();       // Allocate space in memory for Cat
Cat.Species = “Cat”;
Cat.NumberOfLegs = 4;
Cat.Weight = 4.7;

Animal Rover = new Animal();
Rover.Species = “Dog”;
Rover.NumberOfLegs = 4;
Rover.Weight = 23.9;
```

There are three obvious differences.

- Without going into how C# handles memory management, we have to allocate space for an instance of an Animal (and every other class) via the new keyword. Perhaps not surprisingly, once we’ve allocated memory for a class (this is called instantiating an instance of the class), the instance is called, generically, an *object*.
- By default, all fields inside a class can’t be referred to outside the class; they’re private. So we’ll explicitly mark them as public. We’ll see what the implications of this distinction are later.
- You must qualify which object instance a given field refers to. Cat and Dog refer to two different objects, and Cat.Weight and Dog.Weight refer to two different fields. Note that this is the same as in C.

But, as you can see, a C# class is essentially identical to the simpler C struct. Actually, a class is a superset of a struct, as we’ll see below. But if you want to use a class as an almost one-for-one replacement for a struct, go for it!

The Grand Difference

The most major difference between a C struct and a C# class is that a class can contain *methods*[2]. Why is that a good thing?

Perhaps a good way to look at it is the following. The purpose of a struct/class is to represent a concept. In our simple Animal class, we have three fields – species, weight, number of legs. OK, not the most comprehensive description of what an Animal is. But it only describes what an Animal *is*. It's missing what an Animal *does*!

So let's add a field to our Animal class – public string Sound;. Then we could add

```
Cat.Sound = "Miaow";
Dog.Sound = "Woof";
```

So we could write

```
Console.WriteLine[3](Cat.Sound);
```

OK, that can work. But let's put in something that an Animal can *do*.

```
class Animal {
        public string Species;     // See note on "public" below
        public float Weight;
        public int NumberOfLegs;
        public string Sound;

        public void Speak() {
                Console.WriteLine(Sound);
        }
}
```

So once we've set up the fields in our Cat and Dog variables, we can write

```
Cat.Speak();            // Prints Miaow
Dog.Speak();            // Prints Woof
```

So now we've (somewhat) fleshed out our concept of an Animal. We know its characteristics (weight, species, etc.) and what it can do (Speak).

This is called *encapsulation*. We've got this concept called Animal and it defines (encapsulates) everything it is and can do, all between { a pair of braces }.

Encapsulation is one of the three pillars that Object Oriented Programming (OOP) is based upon. The other two will (hopefully!) be described in other documents.

Personally, I find encapsulation the most important pillar. It helps me understand just what it is that I'm working with, what it can do, and so forth.

---

[2] Or, as they are called in other languages, functions and subroutines.
[3] Roughly comparable to C's printf.

While I've described methods as what an object (like an Animal) can do, methods aren't limited to that. To take an admittedly silly example, suppose you needed to calculate the weight supported by each foot. Simple enough.

```
        public float WeightPerFoot() {
                return Weight / NumberOfFeet;
}
```

Side note: For simple methods, C# recently introduced something they call Expression-Bodied members. You can simplify the above by writing

```
        public float WeightPerFoot() => Weight / NumberOfFeet;
```

Public vs. Private

I mentioned earlier that any fields you have inside a class, by default, are private, meaning that they *can't* be used outside the class. I originally defined Animal's fields as public, so we could.

But our Speak() method would have worked even if the Sound field were private. That's because it's referred to *inside* the class, in the Speak() method which in turn is inside the class.

As I've discussed elsewhere, global variables and functions are frowned upon (and C# doesn't allow them), since it restricts one member of a team from changing the specs or implementation details of his routines (e.g. fields and/or algorithms) in case they affect usage of them by other members of the team.

There's a school of thought that says that fields within a class should never be public, unless the programmer guarantees that they'll never change. This includes, for example, units. Our Weight field doesn't say whether it represents pounds or kilograms. Or ounces or grams. And so on. So the programmer doesn't have the flexibility (for whatever implementation reasons) to change the meaning of Weight (say from pounds to kilograms), even though its data type (float) remains the same. But if he did, then other users of this field would find that any calculations done with this public field were now out by a factor of 2.2!

Now in small personal programs, I often[4] don't bother and declare most fields public[5]. But if I declared all fields private, how could I set the internal fields (Species, etc.)? See the next section.

Constructors

A *constructor* (often called a *ctor* (see-tor)) is a method that has the same name as the class it's in. For example,

```
class Quadruped {
        int NumberOfLegs;       // Defaults to private

        public Quadruped () {
```

[4] OK, OK – usually!

[5] Methods are a different matter. Much of the time, a method is used merely as an internal subroutine to a public method. Those I let default to private.

```
            NumberOfLegs = 4;
        }
}
```

When you write new Quadruped(), C# implicitly calls the ctor. And now we've set the private field NumberOfLegs to a value.

Constructors can have parameters. For example,

```
class Quadruped {
        string Species;         // Defaults to private
        int NumberOfLegs;       // Defaults to private

        public Quadruped () {
            NumberOfLegs = 4;
        }

        public Quadruped(string kind, int nLegs) {
            Species = kind;
            NumberOfLegs = nLegs;
        }
}
```

Fields and State

Here's another way of looking at classes. Forget programming for a minute. In normal language, a person could be in many different states depending on the state of various components of himself. For example…

- Age, Gender, Weight, Height, Address, Phone(s), Birthday, Left-Or-Right-Handed, Number-of-Current-Mistresses (or -Misters), etc, etc, etc – the usual suspects.
- HungerLevel – From Famished to No-Thanks-I-Couldn't-Eat-Another-Bite, with several states in between. (Alternatively, how hungry you are on a scale from 0 to 10.)
- StressLevel – with sub-categories of Work-Related, Money-Related, Family-Related and so on.
- EnergyLevel – from Bright-Eyed-And-Bushy-Tailed to Gawd-I-Need-A-Nap.

Or a car…

- MakerModel/Year – e.g. a '53 DeSoto sedan
- CurbWeight
- GasTankCapacity
- Gas Level – Full to Running-on-Fumes
- DateOfNextStateInspection
- OdometerReading
- DateOfLastOilChange
- ManufacturerSuggestedMilesBetweenOilChanges

- SpeedometerReading
- BreakDepressed – true or false, or maybe how much pressure is being applied to the breaks

And so on. So imagine that whatever fields you use to model a person *define* the overall characteristics and *state* of a person / car / whatever,

TODO:

Making private Fields Available Outside the Class – Properties

So what if we want to make the Weight field available to code outside the class? Well, we could just add the public keyword. But that has its own problem. This would allow anyone to change its value at any time. Which might be bad if this field needed to be correlated to one or more other fields.

For example, while our WeightPerFoot() method is trivial, suppose it were a much more complicated method involving loading and scraping multiple web pages, along with database references. If it were called once, that would be acceptable. But if it were called multiple times, this could impact the performance of our app. So maybe inside the class's constructor it calculated the value once and saved it away in a field, and the WeightPerFoot() method could just return that field. And a hypothetical set_Weight() method could set the Weight field and do the calculations again. But just unilaterally changing Weight directly wouldn't do the related calculations.

So we could make Weight private, and require the user to call a set_Weight() method. That works, but is a bit cumbersome. C# provides an alternative called Properties.

Let's change our Animal class so that public int Weight becomes private int _Weight. Well, OK. We can set the Weight in a constructor but can't directly change it afterwards[6]. But what if wanted to?

Let's consider a related issue first. In the original version of Visual Basic (not Visual Basic.Net), you could make a screen widget (e.g. a pushbutton called MyButton) appear by setting MyButton.Visible = True. Or disappear via MyButton.Visible = False. But how could simply setting a Boolean value in memory show/hide a widget? The answer is that VB knew that when you referred to the Visible field of a widget, some special processing needed to be done behind the scenes. This worked. But this was a special case for certain properties of certain types of objects.

[6] Which might actually be a good thing. In these days where many/most CPUs have multiple cores, having more than one thread of execution simultaneously running can be useful from a performance point of view. Experience has shown that while it's possible to coordinate multiple threads modifying the same field(s), it's cumbersome and error prone. Whereas if a class's fields are immutable (never changing once an instance of the class is created), then certain problems don't appear. And if you want to work on an instance with different values, just create a new instance with the new value(s) and have a new thread process that.

C# makes that facility available to all classes.

Let's strip down our Animal class to contain just a Weight field/property.

```
class Animal {
        private int _Weight;      // A field. The "backing field" for the Weight property
        public Animal(int w) => _Weight = w;
        int Weight {              // A property
                get { return _Weight; }  // Really a method called get_Weight()
                set { _Weight = value; } // Really a method called set_Weight()
        }
}
```

The Weight property (indicated by the opening brace after the datatype/fieldname) is *not* a field. It has no data directly associated with it. Its data is held in a "backing field", in this case called _Weight (it's a common convention that the backing field is the name of the property preceded by an underscore; but that's just a convention. Call it anything you like).

What a property really is, is a pair of blocks of code, one for retrieving the value associated with the property (the get block), and one for modifying it (the set block).

The blocks of code can do anything a normal method can, since (to be compatible with other .Net languages that don't know about properties) they're really just syntactic sugar for methods called get_Weight() and set_Weight(). So assuming our Animal class now has the private _Weight field and the public Weight property, under the hood the C# compiler translates

```
Animal Cat = new Animal(4.7);   // Assume a ctor with only a Weight parameter
int wt = Cat.Weight;
```

into

```
Animal Cat = new Animal(4.7);   // Assume a ctor with only a Weight parameter
int wt = Cat.get_Weight();
```

Note: Originally you had to explicitly declare a backing field (_Weight). But later versions of C# allowed the compiler to implicitly create a backing field for you. So you might see something like

```
int Weight { get; set; }
```

In this introductory essay I won't get into exactly when you might use this.

A Few Examples of Properties – Visual Basic's .Visible

Let's start with the Visual Basic problem of MyButton.Visible = True. This isn't the place to go into what Operating System dependent subroutine calls are required to modify the graphical interface. But let's assume that there's a class Button (which MyButton is an instance of) with an integer field called WidgetID, set when the class was instantiated. Also assume there's are system routines called IsVisible() and SetVisible() that do the obvious thing. So the code would be

```
class Button {
        private int WidgetID;
```

```
        // All kinds of other fields/properties/methods not shown here
        public Button() {
                WidgetID = call to some routine that generates a unique ID;
        }
        public bool Visible {
                get { return IsVisible(WidgetID); }
                set { SetVisible(WidgetID, value); }        // See note on value below
        }
}
```

Re *value* in the set block: I consider this to be a minor flaw in the design of C# -- *value* kind of looks like a keyword, but it's not. It's just that the parameter to the set_xxx() method is always called *value*.

But now we can write readily write

```
        Button MyButton = new Button();
                MyButton.Visible = false;
                bool IsVis = MyButton.Visible;
        }
```

A Few Examples of Properties – Pounds vs. Kilograms

So let's suppose the Weight field is interpreted in the constructor as pounds, and you later realize that any references to that field should report the data in kilograms.

Now the obvious way to do this is, in the ctor, to just divide the Weight field by 2.2 and you're done. But technically you could write the following property:

```
        public Weight {
                get { return _Weight / 2.2; }
        }
```

Of course this is inefficient, requiring you to do a division every time you reference the Weight field, rather than just doing it once in the ctor. But you get the idea.

A Few Examples of Properties – Files

Without getting ahead of ourselves and discussing how .Net does file programming, imagine a File[7] class with a NextLine property. Assuming appropriate housekeeping fields, constructors, etc, the following code would list the contents of the file:

```
        class File {
                // All kinds of fields, ctors, etc – not shown
                public string NextLine {
                        get { return code to return either the next line, or null on EOF; }
                }
        }
```

[7] The .Net Framework already has a File class, so treat this section as just a proof-of-concept.

```
File MyFile = new File("foo.txt");
string line;
while ((line = MyFile.NextLine) != null) {
        Console.WriteLine(line);
}
```

A Few Examples of Properties – Debugging

Perhaps my favorite non-obvious use of properties is in debugging. Suppose you have a large program with a class that has an Age field in it. And occasionally that field is erroneously set to a negative number. But it's set so many times from so many places under so many circumstances. How would you track down where it's set to the invalid value. Properties to the rescue!

```
class MyClass {
        int _Age;
        public int Age {
                get => _Age;
                set {
                        if (value < 0) {
                                // set breakpoint here, or log it, or something
                        } else {
                                _Age = value;
                        }
                }
        }
}
```

Indexers

There's a variant of Properties called Indexers. Normally you'd use an index to reference individual elements of an array: foo[n].

Again, ignoring how .Net does database access, suppose we had a database with a StockData table, keyed by its stock symbol (e.g. "MSFT", "IBM", "GE", etc). Assume each row in the table matched the following class definition:

```
class Stock {
        public string   StockSymbol;        // e.g. "MSFT"
        public string   FullName;           // e.g. "Microsoft Corporation"
        public float    CurrentPrice;       // 101.16 as of 2018/7/9
        // Undoubtedly other fields
}
```

Now imagine that we had classes that allowed us to write the following:

```
Database db = new Database("My database name");
string SqlQuery = "SELECT * FROM TheMarket WHERE StockSymbol = 'MSFT'";
```

```
Stock MsftData = db.DoQuery(SqlQuery);
```

OK, that can work. But let's enhance our Database class with an indexer:

```
class Database {
    // All the usual suspects – fields, ctors, methods, etc
    public this[string Sym] {
        get {
            string qry = $"SELECT * FROM TheMarket WHERE StockSymbol = '{Sym}";
            return DoQuery(qry);    // Assume DoQuery is a method here
        }
    }
}
```

But first, I must point out that constructing the SQL statement this way is **extremely** unsafe! For an explanation of why that is, see https://en.wikipedia.org/wiki/SQL_injection

The indexer above (identified by this[Type Name] will now let us write:

```
Stock MsftData = db["MSFT"];            // Assumes <db> previously set up
```

So simple! And with a suitable set block, how 'bout writing a new record to the table with

```
db["LRS"] = new Stock("LRS", "Larry Smith", 999.999);
```

Please note that I haven't provided any error detection/handling. What if there's already a "LRS" symbol? And the Database class as stated is good for only one table ("TheMarket"). But you get the idea.

The data type that goes into the definition of the indexer ([string Sym] above) doesn't have to be a string. For example, .Net has a DateTime class that contains a date and a time (duh!). You could use this to get a weather forecast (LocalForecast[new DateTime(2018, 7, 20)]) or perhaps ThisDayInHistory[new DateTime(1776, 7, 4)]). Suppose you have a class Person.

```
var SomeGuy = new Person(/* whatever you need here */);
var Info = RapSheet[SomeGuy];
```

Indexers can, in many cases, simplify your code.

Summary

C# classes are data structures that encapsulate a concept, what it is (fields), what it does and perhaps other calculations (methods). They support information hiding (the private keyword) and provide features to simplify your code, such as properties and indexers. The ultimate goal is to make it more feasible to understand and to work with higher-order concepts.