A Really Simplified Introduction to Object Oriented Programming Using C#

Disclaimer: I've taken a few liberties below with the C# runtime routines (notably the ones that do file I/O), replacing them with pseudocode, so as not to get bogged down in the details of the .Net Framework and distract from the main points of this essay.

Introduction

Suppose you want geographic and demographic data for countries. To keep things as simple as possible, assume you want to keep just three pieces of data per country:

- Its name (e.g. Canada)
- Its population
- Its area

You also want to get statistics about a country. Again to keep things as simple as possible, suppose all we want is population per square mile.

We can do this in C# in a non-OOP manner. We could define three arrays:

```
string[]   CountryNames;
int[]      CountryPopulations;
int[]      CountryAreas;
```

And then we could define a function

```
double PopulationDensity(int n) {
      return CountryPopulations[n] / CountryAreas[n];
}
```

Problems

OK, this works. Sorta. But it does have a number of potential problems. All are small (since this is such a simple example), but hopefully you can imagine how the problems could proliferate in a larger situation. The problems include…

1. Ideally the three arrays should be declared in the same place. But if/when you wanted to add other characteristics (perhaps capital city, average lifespan, GNP, and/or creation date (e.g. for the US, 1776)), then there's nothing to stop a programmer from defining these in a totally different place, even in a different source module!

2. And who knows where the PopulationDensity function lives? And how would you find it among all the other functions in all the source modules? Ditto for any other functions.

3. You'd better make sure that all the arrays are allocated to have the same length. Again, ideally this allocation should be done in one place, but there's no way to enforce this. And everything had better be in sync. It would be bad news if, say the code that initialized the country name vector filled in the data in a different order from the other two vectors!!!

4. The three (or more) arrays are writable from any place. Suppose you wanted to know the areas of, say, Canada, the US and Mexico in square kilometers instead of in square miles. You could declare an array with three elements called CountryArea[]. But then you could have a slip of the finger and write CountryAreas[1] = CountryAreas[5] * 1.6 * 1.6; Bingo, you've just modified the overall area of Canada by referencing CountryAreaS, not CountryArea.

    - Side note: One way to address this concern would be to *not* use simple integers to index both vectors, but to use Enumerated values (the *enum* keyword). But that's a subject for another essay.

Solution to Problem 1

So what can we do to minimize these problems? Well, in C we could use a *struct*. C# also has *struct*s so we'll start with that.

```
struct Countries {
      string[]   CountryNames;
      int[]      CountryPopulations;
      int[]      CountryAreas;
      // Add other vectors here
}
```

So this addresses point 1. All the arrays are inside a single *struct*.

Now before we go any further, I'm going to abandon the *struct* keyword and replace it with the *class* keyword.

From our introductory point of view it's close enough to say that, in C#, a *class* is identical to a *struct* except that by default everything inside is *private* instead of *public*. More on what those keywords later.

So now we have

```
class Countries {
      string[]   CountryNames;
      int[]      CountryPopulations;
      int[]      CountryAreas;
}
```

Solution to Problem 2

Point 2, associated functions, is simple. We place the definition of the routine *inside* the class.

```
class Countries {
      string[]   CountryNames;
      int[]      CountryPopulations;
      int[]      CountryAreas;
      // Add other vectors here

      double PopulationDensity(int n) {
            return CountryPopulations[n] / CountryAreas[n];
      }
}
```

We're beginning to see the OO concept of *encapsulation*. We have this concept of a set of countries, and everything intrinsic to this concept is all wrapped up with a pretty blue bow. Also if we tried to reference the PopulationDensity function outside the context of class Countries, you couldn't. The compiler would give you a compile-time error.

Solution to Problem 3

To address point 3 about initialization, C# has the concept of a *constructor* ("ctor"). Whenever space is allocated to a class, you can specify a routine to be called to initialize the class instance. Add to the class a method *with the same name as the class*. Whenever the class is instantiated (i.e. a new instance of the class has been created), this routine will run. Let's assume we have a file[1] that has the data, one line for each country. So we'll insert the following method into our class…

```
      public Countries() {       // Constructor
            // Remember, all file I/O is just pseudocode
            File f = OpenFile("CountryInfo.txt");
            int nCountries = f.CountNumberOfLinesInFile();
            // Allocate space for our arrays. Note that we
            // effectively guarantee that all the vectors
            // will be the same length
            CountryNames = new string[nCountries];
            CountryPopulations = new int[nCountries];
            CountryAreas = new int[nCountries];
            for (int n = 0; n < nCountries; n++) {
                  f.Read(CountryNames[n], CountryPopulations[n],
CountryAreas[n]);
            }
      }
```

[1] Or a database, or an XML file, or whatever…

Public vs Private

Let me tell you a personal anecdote. Years ago, I worked on a project that was written in traditional C, and designed to be implemented in three stages. When we finished Phase 1, we brought a few more programmers in to work on Phase 2, including a guy named Sam. They, of course, had to come up to speed on what we original programmers had done in Phase 1.

My part of the job had been to design and implement all the utility subroutines we needed (e.g. buffer management). One day I was walking by Sam's desk and noticed that he was looking at the source for the buffer headers. Since it was my code, I asked, purely out of curiosity, why he had source for the headers on his screen. He explained that he planned to write a routine that wanted to use the information in the headers. *Red Alert!* It turned out that we team leaders had decided that we needed a more sophisticated buffer management routine for Phase 2 and that I'd have to significantly modify the headers. Of course Sam knew nothing of this.

The good news is that I was able to tell Sam we were going to change things, and that he couldn't rely on the Phase 1 data structure, and I wound up writing his proposed routine myself, incorporating it into my Phase 2 design. But it was pure chance that I noticed him looking at the buffer headers as I passed by his desk. Otherwise we wouldn't have caught this problem until much later, when I finally modified the headers and it broke his code.

The problem was that the various fields, methods and algorithms used by (in this case) the buffer routines didn't belong to him and (arguably) he shouldn't care. *Or have access to!* As long as the "published" (i.e. committed to be available to the user of the class[2]) methods still worked, the internals shouldn't matter.

So for all fields and methods, C# allows you to declare them either *public*, visible to the programmer, or *private* (the default for classes), so the programmer will get a compile-time error if he tries to reference them.

Since we were programming in C (not C#, C++, etc), we couldn't declare certain fields/methods off-limits. The closest we could have come would have been to place comments in the code, perhaps

```
/* private */ int BufSize;
```

So let's fix our code a bit

```
class Countries {
      public string[]  CountryNames;
      public int[]     CountryPopulations;
      public int[]     CountryAreas;
      // Add other vectors here
```

[2] As opposed to the owner of the class modifying its internals

```
        public double PopulationDensity(int n) {
              return CountryPopulations[n] / CountryAreas[n];
        }

        public Countries() {
              // Remember, all file I/O is just pseudocode
              File f = OpenFile("CountryInfo.txt");
              int nCountries = f.CountNumberOfLinesInFile();
              // Allocate space for our arrays
              CountryNames = new string[nCountries];
              CountryPopulations = new int[nCountries];
              CountryAreas = new int[nCountries];
              for (int n = 0; n < nCountries; n++) {
                    f.Read(CountryNames[n], CountryPopulations[n],
CountryAreas[n]);
              }
        }
}
```

Inheritance and Polymorphism

What we've addressed so far is known as Encapsulation – gathering together everything to do with our concept of countries.

OOP has two other pillars called *Inheritance* and *Polymorphism*. These are more advanced features and are beyond the scope of this (hopefully) elementary article.

Modifying the Design a Bit

OK, but we're still not done. While I suppose that for some applications, the concept we need is that of Countries (plural), it's more likely that we'd want to drill down and have as our basic concept that of an individual country. So let's rewrite the above as…

```
class Country {
      // These fields are deliberately allowed to default
      // to private
      string _Name;
      int _Population;
      int _Area;
      // Add other characteristics here

      public Country(string Name) {    // Constructor
            _Name = Name;
            // Retrieve the data for this country (e.g.
            // Population, etc) from,
```

```
            // say, a database and save it into the _Population
            // and _Area fields
        }

        public double PopulationDensity() {
            return _Population / _Area;
        }

        // See following section on Properties
        public string Name {
            get { return _Name; }
        }

        public int Population {
            get { return _Population; }
        }

        public int Area {
            get { return _ Area; }
        }
    }
```

Other C# Features

When I first wrote this document I was going to put several other aspects of C# here including, among others, Properties and the var keyword. But I later wrote another document. So any features I mention in the rest of this document should appear in the document entitled *An Introduction to Object Oriented Programming in C#*.

OK, Let's Finish This Off Quickly

… since most of this is in the main document.

- Don't create a class with all the country arrays in it. Create a class that represents one country at a time. So this addresses problems 1 and 2.
- Then you can create an array (linked list, whatever) that contains the instances of class Country. This solves problem 3.
- Defining a property (say, *MetricArea*) that returns the area in square kilometers solves problem 4.

One more suggestion. For ease of use, don't define PopulationDensity as a function that requires you to know the order the countries are in (i.e. the parameter *n*), just make it a property of class Country as follows:

```
class Country {
      // all the previous stuff

      public double PopulationDensity {
            get { return _Population / _Area; }
      }
      // Note the lack of a set clause means you
      // can't assign to PopulationDensity
}
```