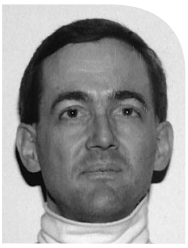
using C# properties and static members

by Glen McCluskey



Glen McCluskey is a consultant with 20 years of experience and has focused on programming languages since 1988. He specializes in Java and C++ performance, testing, and technical documentation areas.

glenm@glenmccl.com

In this column we're going to continue our examination of the C# programming language, and look at two particular features of C# classes.

We'll start by considering the use of properties, which are kind of a hybrid between data fields and methods. We'll then go on and look at static class members.

Properties

Imagine that you're developing a C# class, and that class will have a data field that represents a calendar year. The field is set by the constructor and validated to ensure that the year is 1800 or later. You also want the ability to change the year after the fact, in existing objects of the class.

Here's some C# code that illustrates this approach:

```
using System;

public class Prop1 {
    private int year;

    public Prop1(int y) {
        SetYear(y);
    }

    public int GetYear() {
        return year;
    }

    private const int MINYEAR = 1800;

    public void SetYear(int y) {
        if (y < MINYEAR)
            throw new ArgumentException("year < " +
                    MINYEAR);
        year = y;
    }
}

public class TestProp1 {
    public static void Main() {
        Prop1 p = new Prop1(1956);
        Console.WriteLine("year #1 = " + p.GetYear());

        //p.year = 1977;
        p.SetYear(1977);
        Console.WriteLine("year #2 = " + p.GetYear());
    }
}
```

The year field is private, meaning that it cannot be set directly from outside of the class (see the commented line in Main). If the field is made public, then there's no way to validate a new value that is set. The field is instead changed via the SetYear method, and the proposed new value is checked in this method.

This approach is very common and works pretty well, but it's a little tedious to use, with every private field requiring a pair of get/set access methods.

C# offers another approach to solving this problem, using what are called properties. A property looks like a data field in an object, but access is controlled via internal get/set methods. The property can be made public and accessed like a field, but there is a layer of control that allows the class designer to interpose specific processing when the property is accessed.

Let's look at an example:

```
using System;

public class Prop2 {
    private int year;

    private const int MINYEAR = 1800;

    public int Year {
        get {
            return year;
        }
        set {
            if (value < MINYEAR)
                throw new ArgumentException("year < " +
                        MINYEAR);
            year = value;
        }
    }

    public Prop2(int y) {
        Year = y;
    }
}

public class TestProp2 {
    public static void Main() {
        Prop2 p = new Prop2(1956);
        Console.WriteLine("year #1 = " + p.Year);
```

```
        p.Year = -1977;
        Console.WriteLine("year #2 = " + p.Year);
    }
}
```

There's still a private year field in this code, but also a public property "Year." The property can be thought of as a "virtual field." It's treated like a data field when you're programming with the class, but there are hooks such that the class can control what happens when the property's value is retrieved or set.

In the example above, when the property value is retrieved, the private year field's value is returned. When the property is set, the proposed new value, represented by the keyword value, is first checked to ensure that it's at least 1800. Then the private field is set.

Properties enable simple field access from a programmer's perspective while, at the same time, supporting data hiding so that access to private data can be controlled.

Global Variables and Static Members

C# requires that all data fields be part of a class. This restriction leads to an obvious question: How do you implement global variables, variables that can be accessed from anywhere in your application? Using such variables is not always a good idea, but we're going to assume that you really do want them for some purpose.

To answer this question, we need to consider what is meant by the concept of static class members. Normally, you define a class and then create instances or objects of the class. For example, a Point class that represents X,Y points will have various instances, such as one that represents the point 25,35. The X,Y values in the instance are called instance members.

A static member can be thought of as belonging to the class itself, and not to its instances. For example, a static data field is shared across all instances of a class. There may be one or a million instances of the class in a running application, but there will still be only one copy of the static data for the class.

Static members can be used to implement the equivalent of global variables. Here's an example:

```
// file #1

public class Globals {
    private Globals() {}

    public static int glob1 = 100;
    public static int glob2 = 200;
}

// file #2

using System;

public class TestGlobals {
    public static void Main() {
        //Globals g = new Globals();

        Globals.glob1 = 500;
        Globals.glob2 = 600;

        Console.WriteLine("glob1 = " + Globals.glob1);
        Console.WriteLine("glob2 = " + Globals.glob2);
    }
}
```

Globals is a class with two public static data members. They can be referenced by qualifying the member names with "Globals." Globals also has a private constructor, which cannot be called from outside the class. Defining a private constructor means that no instances of the class can be created. In other words, the class is used simply as a packaging vehicle for static data members.

This same approach can be used for packaging static methods, such as methods that represent self-contained mathematical functions and that have no meaning as conventional methods that operate on class instances. Let's look at an example:

```
using System;

public class CircleFuncs {
    private CircleFuncs() {}

    public static double GetCircumference(double r) {
        return 2.0 * Math.PI * r;
    }

    public static double GetArea(double r) {
        return Math.PI * r * r;
    }
}

public class TestCircleFuncs {
    public static void Main() {
        double radius = 10.0;

        Console.WriteLine("circumference = " +
            CircleFuncs.GetCircumference(radius));
        Console.WriteLine("area = " +
            CircleFuncs.GetArea(radius));
    }
}
```

CircleFuncs is a class that groups together some methods used to calculate properties of a circle, such as its circumference and area.

Note that the Main method, the entry point to a C# application, is static. It's part of a class – in the example above, the class TestCircleFuncs – but it doesn't operate on instances of TestCircleFuncs.

Constants

Constants are closely related to static members. If you're doing C# programming, how do you define groups of constants for use in your program? Let's look at a couple of examples that illustrate some of the techniques that are available:

```
using System;

enum Color {RED = 1, GREEN = 2, BLUE = 3}

public class Const {
    private Const() {}

    public const string RED = "red";
    public const string GREEN = "green";
    public const string BLUE = "blue";
}

public class TestConst {
    public static void Main() {
        Console.WriteLine("Color.GREEN = " +
            Color.GREEN);
        Console.WriteLine("Color.GREEN as int value = " +
            (int)Color.GREEN);
        Console.WriteLine("Const.GREEN = " +
            Const.GREEN);
    }
}
```

In this first example, we use an enumerated type, very similar to what C and C++ offer. This approach works as you would expect, but is suitable only for integral types.

The Const class shows how to define a group of string constants. A private constructor is once again used, so that no instances of the Const class can be created. The fields are marked as Const, which means that they're static and cannot be changed after initialization.

When you run this program, the output is:

```
Color.GREEN = GREEN
Color.GREEN as int value = 2
Const.GREEN = green
```

Here's another slightly more complicated example:

```
using System;

public class Primes {
    public const uint NUMPRIMES = 10;
    public static readonly uint[] PRIMES;

    private Primes() {}

    private static bool IsPrime(uint p) {
        if (p <= 2)
            return p == 2;
        if (p % 2 == 0)
            return false;
        uint last = (uint)Math.Sqrt(p);
        for (uint i = 3; i <= last; i += 2) {
            if (p % i == 0)
                return false;
        }
        return true;
    }

    static Primes() {
        PRIMES = new uint[NUMPRIMES];
        uint currvalue = 1;
        for (uint i = 0; i < NUMPRIMES; i++) {
            while (!IsPrime(currvalue))
                currvalue++;
            PRIMES[i] = currvalue++;
        }
    }
}

public class TestPrimes {
    public static void Main() {
        Console.Write("primes = ");
        for (uint i = 0; i < Primes.NUMPRIMES; i++)
            Console.Write(Primes.PRIMES[i] + " ");
        Console.WriteLine();
    }
}
```

In this example, we want to compute a table of prime numbers. We want the table to be constant and thus not mutable after it's initialized, but at the same time, we'd like to compute the values in the table at runtime, rather than actually listing them out (2, 3, 5, 7, 11, ...) in the source code.

There are a couple of techniques that we use to implement this approach. We mark the PRIMES array as static and read-only. The read-only qualifier means that the array can be modified in the constructor, but not afterwards.

We also use a static constructor, which is called when the static data members for the Primes class are initialized. The class has both an instance constructor, which is private and used to disallow creation of class instances, and a static constructor, used to initialize the PRIMES field.

The output of this program is:

```
primes = 2 3 5 7 11 13 17 19 23 29
```

In future columns we'll start looking at some broader issues with classes, such as programming with interfaces and abstract classes.