;login:
THE MAGAZINE OF USENIX & SAGE
August 2003 • volume 28 • number 4

inside:

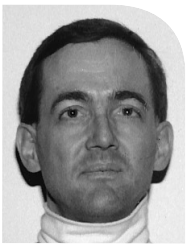
PROGRAMMING

McCluskey: Working with C# Classes

USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

# working with C# classes

**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 previous columns, we've looked at some of the basics of the C# language. It's now time to start examining in more detail how C# classes work. A class is a user-defined type, and serves as the fundamental unit of design and composition for C# programs.

## A Class to Represent X,Y Points

Let's start our discussion by looking at a class whose instances represent X,Y points. Previously we defined a class as being a combination of some data (think of a C struct) plus operations that manipulate instances or objects containing that data. For a Point class, the data would likely be a couple of integers representing the point (like 25,100), along with operations to initialize a point object, access the X,Y values in an object, compare an object to other point objects, convert an object to a string for printing and formatting, and so on.

Here's some C# code that illustrates these ideas:

```
using System;

public class Point {

    private int x, y; // X,Y data fields

    // constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // copy constructor
    public Point(Point p) {
        this.x = p.x;
        this.y = p.y;
    }

    // accessor methods
    public int getX() { return x; }
    public int getY() { return y; }

    // conversion to string

    public override string ToString() {
        return String.Format("({0},{1})", x, y);
    }

    // equality check against another Point object
    public override bool Equals(object obj) {
        if (!(obj is Point))
            return false;
        Point p = (Point)obj;
        return x == p.x && y == p.y;
    }

    // hash code for the object
    public override int GetHashCode() { return (x << 16) | y; }
}

public class Test {
    public static void Main() {

        // create some Point objects on the heap:

        Point p1 = new Point(50, 75);
        Point p2 = new Point(50, 75);
        Point p3 = new Point(100, 150);
        Point p4 = new Point(p3);// copy constructor for Point
        // access the X,Y values of an object

        Console.WriteLine("p1 X,Y = {0},{1}", p1.getX(), p1.getY());

        // convert object to string and print it

        Console.WriteLine("p4 = {0}", p4);

        // exercise the overridden Equals() method

        if (p1.Equals(p2))
            Console.WriteLine("p1/p2 equal");
        if (p2.Equals(p3))
            Console.WriteLine("p2/p3 equal");
        // get the hash code for an object

        Console.WriteLine("p1 hash code = {0}", p1.GetHashCode());
    }
}
```

The Point class defines two data fields to hold the X,Y values. These are private fields, which means that only methods of the Point class can access the fields. In particular, it's not legal to say:

```
Point p = new Point(10, 20);
p.x = -125;
```

If this kind of operation is allowed, then the internal representation details of a Point object are exposed to the user, generally an undesirable thing (especially if the representation changes at a later time). Also, allowing the field to be set directly may violate the integrity and domain of the object. For example, if a

check is made that X,Y are positive integers when the object is created, then setting the X value to -125 might cause havoc.

The first two methods of the Point class have the same name as the class itself (Point), and thus are constructors, special methods used to initialize Point objects. The memory for objects is allocated from the heap, and the constructor is called to initialize the raw memory.

The second constructor is a copy constructor, used to make a copy of an already existing Point object. In lieu of using such a constructor, we could instead say:

```
Point p1 = new Point(10, 20);
Point p2 = new Point(p1.getX(), p1.getY());
```

but a copy constructor is a more general mechanism. Note that a byte-by-byte copy of an object is rarely the right choice, given that an object may contain internal references to other objects.

getX and getY are accessor methods, used to access the private X,Y fields in Point objects.

ToString, Equals, and GetHashCode are methods found in the root class (System.Object), and overridden in Point to provide custom behavior.

For example, the method System.Object.ToString has certain generic behavior, illustrated in this example that defines a dummy Point class:

```
using System;

public class Point {}

public class Test {
      public static void Main() {
          Point p1 = new Point();
          Console.WriteLine(p1);
          Point p2 = new Point();
          Console.WriteLine(p2);
      }
}
```

When Console.WriteLine is called, its argument (p1 or p2) must be converted to a string for printing; the method System.Object.ToString is used for this because we didn't override ToString in the dummy class. The default ToString behavior uses the name of the class itself as the value returned by ToString, so the result of running this program is:

```
Point
Point
```

For real classes, ToString should have per-object customized behavior, by including, for example, the distinct X,Y values found in a Point object.

The same consideration applies to the Equals method, used to compare two Point objects for equality. Such equality checking needs to be more sophisticated than merely doing a binary comparison of the corresponding bytes in the two objects. For example, objects may contain references to sub-objects, and comparing the memory pointers of the sub-objects will not work.

GetHashCode is used to compute a hash code for an object. The hash code is used by collection classes that represent groups of objects, such as a hashtable.

When the Point class demo is executed, the output is:

```
p1 X,Y = 50,75
p4 = (100,150)
p1/p2 equal
p1 hash code = 3276875
```

### Other Ways to Implement the Point Class

There are other ways we could define a Point class. For example, we could use a struct instead of a class. A struct is a simpler form of a class, with some restrictions and differences. Struct objects are allocated on the stack instead of the heap, and, as discussed in our previous column, have value rather than reference semantics.

Another possibility is to use a class in a very simple and low-level way:

```
public class Point {
      public int x, y;
      public Point(int x, int y) {
          this.x = x;
          this.y = y;
      }
}
```

This approach is not much different from using a C struct. Doing things this way violates the whole object-oriented paradigm but, at the same time, is a simple approach sometimes useful in casual programming, e.g., when you're building a prototype.

A third alternative is to use C# properties. A property is kind of a cross between a data field and a method. Properties are referenced like data fields, but have get/set methods to control the access.

## Creating and Reclaiming Objects

In the discussion above, we talked about how memory is allocated for class objects, with the class's constructor called to initialize the memory. What happens when an object is no longer in use? How do you get rid of it and reclaim the space?

C# uses automatic garbage collection to reclaim objects, so most of the time you don't need to worry about the details of memory management. What does this mean in practice? Let's look at an example:

```
using System;
using System.Threading;

public class CtorDtor {

    // constructor
    public CtorDtor() {
        Console.WriteLine("constructor called");
    }

    // destructor (actually the finalize method
    // commented below)
    ~CtorDtor() {
        Console.WriteLine("destructor called");
    }

    //protected override void Finalize() {}
}

public class Test {

    // create a CtorDtor object, which
    // becomes garbage when f() exits

    static void f() {
        CtorDtor cd = new CtorDtor();
    }

    public static void Main() {
        f();

        //GC.Collect();
        //Thread.Sleep(500);

        Console.Write("Press Enter to quit program: ");
        Console.ReadLine();
    }
}
```

In this code, the Main method calls f, and f calls new to create an object of the CtorDtor class on the heap. Then f returns, and at this point, there is no way to reference the CtorDtor object created in f, and thus this object has become garbage.

Such garbage is subject to reclamation at any time, but garbage collectors typically run in a separate program thread and try to minimize performance overhead. For example, a garbage collector may run only when free memory is getting low. It's unwise to assume particular garbage collection behavior.

When the garbage collector runs, one of the things it does is call finalize methods on objects. A finalize method is used to perform any cleanup that is required (other than reclaiming space) – for example, freeing up system resources not under the control of the C# runtime system.

C# supports C++ destructor syntax, like this:

```
~CtorDtor() {}
```

but this syntax is actually an alias for:

```
protected override void Finalize() {}
```

Finalize methods are not really the same as destructors. In the C++ world, a destructor is called when an object goes out of scope (such as a stack-based object at function exit) or when the delete operator is called for a heap-allocated object.

By contrast, a finalize method is called by the garbage collector, and it's risky to rely on the garbage collector behaving in a specific way (or running at all).

When the CtorDtor demo program runs, the output indicates that the destructor (finalize method) is not called until the program exits. One way of forcing the finalizer to run sooner is to explicitly call garbage collection, using the commented lines of code. This approach works, but is not recommended. Before you start relying on explicit garbage collection calls, it pays to sit down and really study how garbage collection works – it's a tricky area to make assumptions in.

## Explicitly Disposing of Objects

Garbage collection isn't always effective in calling finalize methods in a timely way. What do you do if you have a class whose objects represent critical system resources, resources that are in short supply? One solution is to implement the IDisposable interface in a class:

```
using System;

public class CtorDispose : IDisposable {

    // a resource, with a value of -1 indicating
    // that the resource is currently unallocated

    private int resource = -1;

    public CtorDispose() {
        // allocate resource
        resource = 100;
    }

    ~CtorDispose() { DoDispose(); }
    void DoDispose() {          // free up resource
        resource = -1;
    }

    public void Dispose() {
        DoDispose();
        GC.SuppressFinalize(this);
    }
}
```

```
public class Test {
    public static void Main() {
        CtorDispose cd = new CtorDispose();
        cd.Dispose();
    }
}
```



An interface like IDisposable declares certain methods, in this case Dispose. A class that implements the interface must define the methods. Implementing IDisposable and defining Dispose means that a class offers a way to directly force object cleanup, without waiting for the finalize method to be called by the garbage collector (the finalize method cannot be called directly, and garbage collection may not occur in a timely way).

In the code above, Dispose is called directly, and the method frees the system resource. The garbage collector is then informed that the object (this) should not be finalized at garbage collection time.

In future columns, we'll look at some further details of how classes work, and how classes and interfaces can be combined to build applications.