;login:

THE MAGAZINE OF USENIX & SAGE

April 2003 • volume 28 • number 2

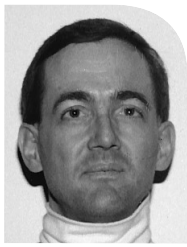
inside:

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

examining the C# "hello, world" program

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 our introductory C# column, we discussed some of the background and context for the C# language, in particular how C# fits into the .NET Framework. In this column we'll start describing the C# language itself.

The Hello Program

Let's look first at the C# version of the "Hello, world" program. This is a trivial application but one we can use to tie down many of the C# basics. Here's the code:

```
using System;

class Hello {
      static void Main() {
          Console.WriteLine("Hello, world!");
      }
}
```

We're going to assume the use of the Microsoft SDK in our discussions. Given this, we compile and execute the program by saying:

```
$ csc Hello.cs
$ Hello
```

The compilation produces an EXE file. This file is very small (3072 bytes) and is not self-contained. It depends on the presence on the local system of the .NET JIT (just-in-time compiler), as discussed in the introductory column. The csc compiler generates opcodes in an intermediate language, and the opcodes are stored in the EXE. When the EXE is executed at some later time, the JIT is used to translate the intermediate language into machine language.

We put our Hello program in a source file Hello.cs. But there's no file-naming requirement implied by doing so; we can instead say:

```
$ cp Hello.cs xyz.cs
$ csc xyz.cs
$ xyz
```

and it will still work.

Another point about this example is that there is a distinguished method named Main() in a C# program, used as the entry point for program execution. Main is a static method in the Hello class, "static method" meaning that the Main method does not operate on instances of the Hello class but is part of the class for packaging purposes. In this particular example, we don't actually create any Hello class objects. For languages like C++ and C#, using a class as a packaging vehicle for static methods and data is a common program-structuring technique.

Classes are a basic unit of composition and design in C#. C# programming consists of the development of new types, realized via classes and the related struct and interface mechanisms, along with use of standard types such as System.String.

The Main method is called to begin execution of the Hello program, and there's a single statement to execute. Console is a class found in the System namespace, and the "using System" statement says that the types found in this namespace are made available to the Hello program. If we got rid of the using statement, we'd need to say:

```
System.Console.WriteLine("Hello, world!");
```

Using statements is very convenient but can sometimes pollute the application with extraneous and conflicting names.

Namespaces are another basic design mechanism for C# programs; they serve to collect and segregate names in a large application. In our example, we refer to the System namespace. There's also a single global namespace to which the Hello type is added, given that we don't specify a namespace for the Hello class. We could instead say:

```
namespace ABC {
      class Hello {
          ...
      }
}
```

to put the Hello class into the ABC namespace.

Actual output from the Hello program is done by the WriteLine method. This is a static method that is part of the Console class found in the System namespace. It writes output to standard output and is equivalent to:

```
Console.Out.WriteLine("Hello, world!");
```

in which the stream (In, Out, Error) is specified. I/O occurs using standard streams, and I can redirect the output in the usual way:

```
$ Hello > out
```

For the statement:

```
System.Console.Out.WriteLine("Hello, world!");
```

System is a namespace, Console a class in that namespace, Out a static stream object of class TextWriter in the Console class, and WriteLine a method in the TextWriter class.

Command-Line Arguments and Exit Codes

Let's go on and look at another program, this one a variation of the Hello program. It illustrates some additional C# features.

Suppose that you'd like to write the Hello program, but instead of the output going to standard output, you want to specify the output file on the command line, and have output written to that file. How can you do this? Here's an example:

```
using System;
using System.IO;

class FileIO {
    static void Main(string[] args) {
        if (args.Length != 1) {
            Console.Error.WriteLine("missing filename");
            Environment.Exit(1);
        }

        try {
            StreamWriter sw = new StreamWriter(args[0]);
            sw.WriteLine("Hello, world!");
            sw.Close();
        }
        catch {
            Console.Error.WriteLine("couldn't open file");
            Environment.Exit(1);
        }
    }
}
```

Command-line arguments are represented by an array of strings. If there's no command-line argument specified, the program writes an error message to standard error and exits with a non-zero status.

Otherwise, the StreamWriter class is used to perform output to a file. Note that StreamWriter operations are wrapped in a try/catch block. This is done because C# uses exceptions to signal error conditions: for example, failure to open a text file for writing. We use the try/catch block to catch any exception that is thrown. If there is no try/catch block, and an invalid file is specified, the program will terminate with an unhandled-exception diagnostic.

Character Encodings

C# uses the Unicode 16-bit character set. However, in our examples so far, we've implicitly assumed that ASCII is being used: for example, when redirecting output to a file. How does this work? C# I/O uses encodings to map Unicode into other character sets. For example, when you run this little program:

```
using System;
using System.IO;

class Encode {
    static void Main() {
        StreamWriter sw = new StreamWriter("out");
        Console.WriteLine("file encoding is: "
                + sw.Encoding);
        sw.Close();
    }
}
```

the result is:

```
file encoding is: System.Text.UTF8Encoding
```

Roughly speaking, the UTF-8 encoding maps Unicode characters with 7-bit ASCII values into the corresponding ASCII characters, and other Unicode characters into two or three bytes. So C# can use Unicode and still be compatible with ASCII. A similar mechanism is used in Java I/O.

Applications with Multiple Source Files

Suppose that you'd like to write the Hello program, but as part of a more elaborate system whereby messages are logged to a file along with a timestamp. How might this be done?

In such a case it is worth defining your own C# class. Instances of the class represent an open file, to which messages are being logged. Here's what the code looks like:

```
// MainFile.cs

class MainFile {
    static void Main(string[] args) {
        FileLogger flog = new FileLogger(args[0]);
        flog.write("Hello, world!");
        flog.close();
    }
}

// FileLogger.cs

using System;
using System.IO;

class FileLogger {
    private StreamWriter sw;

    public FileLogger(string fn) {
        sw = new StreamWriter(fn);
    }

    public void write(string msg) {
        sw.WriteLine(DateTime.Now + ":  " + msg);
    }
```

```
    public void close() {
        sw.Close();
    }
}
```

You compile this program by saying:

```
$ csc MainFile.cs FileLogger.cs
```

Compilation units can depend on each other. For example, the MainFile class uses the FileLogger class, defined in a separate file.

When you run the program, by saying:

```
$ MainFile outlog
```

the result written to the log file is something like this:

```
11/29/2002 10:33:35 AM:  Hello, world!
```

This particular application does not handle exceptions via try/catch. If you specify an invalid log file name, for example:

```
$ MainFile .
```

the program will abort, and display a stack traceback for the unhandled exception.

Browsing System Types

There's one final area we'd like to look at in our discussion of the C# Hello program. Earlier we mentioned the System.Console class, a standard class used for console I/O. How can you know what the standard classes are? Obviously, you can consult reference books, browse online documentation, and look at Web sites.

But there's another way to find out what standard types are available, using the reflection features of C#. You can write a C# program that essentially asks itself what types it knows about and displays the names of those types.

Here's some code that does this:

```
using System;
using System.Reflection;

class DumpTypes {

    // Display a help message and exit.

    private static void dohelp() {
        Console.WriteLine("Usage: [-h|-help] " +
            "[-dumpmem|-m] [-t|-type targ_type] patt1 patt2 ...");
        System.Environment.Exit(0);
    }

    // Filter a string to determine if it contains
    // patterns specified on the command line.

    private static bool filter(string str, string[] args, int argi) {
        str = str.ToLower();

        for (int i = argi; i < args.Length; i++) {
            if (str.IndexOf(args[i]) == -1)
                return false;
        }

        return true;
    }

    public static void Main(string[] args) {
        bool mem_flag = false;
        string targ_type = null;

        // Parse command-line arguments.

    int argi = 0;
        while (argi < args.Length && args[argi][0] == '-') {
            if (args[argi] == "-dumpmem" || args[argi] == "-m") {
                mem_flag = true;
            }
            else if (args[argi] == "-t" || args[argi] == "-type") {
                if (argi + 1 < args.Length) {
                    targ_type = args[argi + 1].ToLower();
                    argi++;
                }
            }
            else if (args[argi] == "-h" || args[argi] == "-help") {
                dohelp();
            }

            argi++;
        }
        for (int i = argi; i < args.Length; i++)
            args[i] = args[i].ToLower();

        // Get a list of all types found in the standard
        // library.

        Assembly asm = Assembly.Load("mscorlib.dll");
        Type[] typelist = asm.GetTypes();

        // Iterate across each type.

        foreach (Type atype in typelist) {

            // Check whether type matches specified
            // target type.

            string typestr = atype.ToString();
            if (targ_type != null && typestr.ToLower() !=
                        targ_type)
                continue;

            // If asked to display all members, iterate
            // across them.

            if (mem_flag) {
                MemberInfo[] memlist = atype.GetMembers();
```

```
                foreach (MemberInfo amember in memlist) {
                    string memstr = amember.ToString();
                    string s = typestr + " " + memstr;
                    if (filter(s, args, argi))
                        Console.WriteLine("{0}   {1}",
                            typestr, memstr);
                }
            }

            // Just display the type itself without
            // members.

            else {
            if (filter(typestr, args, argi))
                Console.WriteLine(typestr);
            }
        }
    }
}
```

The source code for this program and others in this column is available at *ftp://ftp.glenmccl.com/pub/usenix/cs2.zip*.

The heart of the DumpTypes program is these two lines:

```
Assembly asm = Assembly.Load("mscorlib.dll");
Type[] typelist = asm.GetTypes();
```

An assembly is a collection of files something like a shared library or archive. These lines of code load the standard C# assembly and extract a list of types from it, using the GetTypes method. Given a type such as a class and it's possible to call an analogous method (GetMembers) to find a list of the members (methods, fields, properties) defined within the type.

If you run this program with no command-line arguments, it displays a list of standard types:

```
System.Object
System.ICloneable
System.Collections.IEnumerable
System.Collections.ICollection
System.Collections.IList
...
```

If you specify a given type:

```
$ DumpTypes -t System.String
```

it displays just that type. If you specify that members are to be displayed as well, like this:

```
$ DumpTypes -t System.String -m
```

the result is a list of members for the System.String class:

```
System.String  System.String Empty
System.String  System.String ToString(System.IFormatProvider)
System.String  System.TypeCode GetTypeCode()
System.String  System.Object Clone()
System.String  Int32 CompareTo(System.Object)
...
```

For example, the first line of output refers to the Empty static field of System.String. Empty is of type System.String and refers to an empty string ("").

If you specify a list of matching patterns, every line of output will be filtered against all those patterns. For example, saying:

```
$ DumpTypes -m
```

provides a voluminous list of all types and members, a total of more than 20,000 lines of output. But if you say:

```
$ DumpTypes -m cos
```

the output is:

```
System.Math   Double Acos(Double)
System.Math   Double Cos(Double)
System.Math   Double Cosh(Double)
```

which describes three trigonometric methods in the System.Math class.

We've looked at some of the basics around writing C# programs. C# is typically pitched as a language for developing GUI and network applications, but you can also write stand-alone utility programs just as you would in C programming.