GLEN MCCLUSKEY

working with C# serialization

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

glenm@glenmccl.com

AT SOME POINT IN THE DEVELOPMENT of most software applications, design decisions are made about how to store and retrieve application data. For example, if your application reads and writes to disk files, you need to make basic choices about how to represent the data on disk.

In this column we want to look a bit at C# I/O issues, and in particular at a mechanism called serialization. Serialization is used to convert C# objects into bytestreams, in a standardized way, so that those objects can be saved to disk or sent across a network.

The Need for Serialization

Let's start by considering a couple of examples. The first one writes a floating-point value to a text file and then reads it back:

```
using System;
using System.IO;

public class SerialDemo1 {
   public static void Main() {

      // write double value to text file

      double d1 = 0.1 + 0.1 + 0.1;
      StreamWriter sw =
            new StreamWriter("out", false);
      sw.WriteLine(d1);
      sw.Close();

      // read double value back from text file

      StreamReader sr = new
            StreamReader("out");
      string ln = sr.ReadLine();
      double d2 = Double.Parse(ln);
      sr.Close();

      // compare values

      if (d1 != d2) {
            Console.WriteLine("d1 != d2");
            Console.WriteLine("difference = " +
               (d1 - d2));
      }
   }
}
```

When this program is run, the result is:

```
d1 != d2
difference = 5.55111512312578E-17
```

For some reason, our attempt to store a floating value in a text file fails. If we know much about floating-point, we may not be surprised, given that many decimal values have no exact representation in binary. For example, the common value 0.1 is the sum of an infi-

nite series of binary fractions. Somehow our initial value got changed a bit, due to roundoff factors and so forth.

Here's another example. This time we're trying to store short (16-bit) values, and our program is as follows:

```
using System;
using System.IO;

    public class SerialDemo2 {
        public static void Main() {

            // write short value to binary file

            short s1 = 12345;
            FileStream fs1 =
                new FileStream("out", FileMode.Create);
            fs1.WriteByte((byte)((s1 >> 8) & 0xff));
            fs1.WriteByte((byte)(s1 & 0xff));
            fs1.Close();

            // read short value from binary file

            FileStream fs2 =
                new FileStream("out", FileMode.Open);
            int b1 = fs2.ReadByte();
            int b2 = fs2.ReadByte();
            short s2 = (short)((b2 << 8) | b1);
            fs2.Close();

            // compare short values

            if (s1 != s2) {
                Console.WriteLine("s1 != s2");
                Console.WriteLine("difference = " +
                    (s1 - s2));
        }
    }
}
```

Our approach in this code is to pick apart the values, write the individual bytes to a binary file, then read them back.

The output of this program is:

```
s1 != s2
difference = -2295
```

Unfortunately, we made a mistake when we read the value back from the file—we got the byte order wrong.

These examples serve to illustrate why serialization is important—there needs to be some standard mechanism for converting objects into bytestreams and back again.

A Serialization Example

Let's go back to the first example, where we are trying to store a floating-point value. Let's assume that we have such a value represented in an object, and we want to store that object to a file and then later retrieve it. Here's some code that will do so:

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization.Formatters.Soap;

[Serializable]
```

```
public class MyObject {
    private double dvalue;

    public MyObject(double dvalue) {
        this.dvalue = dvalue;
    }

    public double GetValue() {
        return dvalue;
    }
}

public class SerialDemo3 {
    public static void Main() {

            // serialize MyObject instance to file

            double d1 = 0.1 + 0.1 + 0.1;
            MyObject obj1 = new MyObject(d1);
            FileStream fs1 =
                    new FileStream("out", FileMode.Create);
            BinaryFormatter fmt1 = new BinaryFormatter();
            fmt1.Serialize(fs1, obj1);
            fs1.Close();

            // deserialize MyObject instance from file

            FileStream fs2 =
                    new FileStream("out", FileMode.Open);
            BinaryFormatter fmt2 = new BinaryFormatter();
            MyObject obj2 = (MyObject)fmt2.Deserialize(fs2);
            fs2.Close();
            double d2 = obj2.GetValue();

            // compare object values

            if (d1 != d2) {
                    Console.WriteLine("d1 != d2");
                    Console.WriteLine("difference = " +
                            (d1 - d2));
        }
    }
}
```

The example creates an instance of MyObject, containing a double value, and then creates a BinaryFormatter. The formatter is used to convert an object into a bytestream and takes care of all details of conversion, such as traversing arrays and object graphs (for example, linked lists).

At a later point, the process is reversed and the bytestream converts back into an object. Obviously, the serialization process is making use of some internal format for laying out objects and individual data items such as floating-point values.

Other kinds of formatting are possible. For example, if in our demo we go through and change BinaryFormatter to SoapFormatter, then the serialization format will be XML-based.

Using serialization in this way takes care of our original problem with being able to represent floating-point values exactly.

Transient Data

Making use of serialization is often as simple as the previous example illustrates, with all the details handled automatically. But it is also possible to exercise a finer degree of control over the process.

Let's consider another example, where we have an object containing an internal table, a table whose values are computed in some obvious way. By default, the serialization process will convert such tables into a bytestream, along with all other fields in the object. However, doing so may waste a lot of space.

To get around this problem, it is possible to specify that certain fields in an object not be serialized and, further, to specify a callback mechanism that is invoked when an object is deserialized. Using the callback, the object's table can be reconstructed at deserialization time.

Here's what the code looks like:

```
    using System;
    using System.IO;
    using System.Runtime.Serialization;
    using System.Runtime.Serialization.Formatters.Binary;

    [Serializable]
    public class MyObject : IDeserializationCallback {
        private uint length;
        private uint offset;
        [NonSerialized] private uint[] tab;

        private void buildtab() {
                tab = new uint[length];
                for (uint i = 0; i < length; i++)
                        tab[i] = i * i + offset;
    }

    public MyObject(uint length, uint offset) {
                this.length = length;
                this.offset = offset;
                buildtab();
    }

    public uint GetValue(uint index) {
                return tab[index];
    }

    public virtual void OnDeserialization(Object x) {
                buildtab();
    }
}

public class SerialDemo4 {
    public static void Main() {

            // serialize object to file

            MyObject obj1 = new MyObject(1000, 1000);
            FileStream fs1 =
                    new FileStream("out", FileMode.Create);
            BinaryFormatter fmt1 = new BinaryFormatter();
            fmt1.Serialize(fs1, obj1);
            fs1.Close();

            // deserialize object from file

            FileStream fs2 =
                    new FileStream("out", FileMode.Open);
            BinaryFormatter fmt2 = new BinaryFormatter();
            MyObject obj2 = (MyObject)fmt2.Deserialize(fs2);
            fs2.Close();
            Console.WriteLine(obj2.GetValue(25));
    }
}
```

The internal table in this example is a pretty simple one, a table of squares plus an offset. For example, the value for 25 will be 1625, or 25 * 25 + 1000. Since the table can be computed, there's no point in serializing it. Instead, we rely on the OnDeserialization method as a hook to be called when an instance of MyObject is deserialized.

In other words, only the table length and offset are serialized for MyObject instances, and these values are sufficient to reconstruct the object. The actual table is marked with a [NonSerialized] attribute. Doing it this way saves a great deal of disk space. More generally, you might wish to use a scheme of this sort when the physical and logical representations of an object are fundamentally different from each other.

Serialization is an important tool that you can use to store and transmit C# data in a standardized way. You no longer have to worry about devising custom data formats or about issues such as byte ordering.