Internet—and local area network—services are built. HTTP, FTP, and SMTP use TCP; DNS uses UDP. TCP is connection-oriented and includes reliability mechanisms; UDP is connectionless, has a lower overhead, and supports broadcasting. *BitTorrent* uses UDP, as does Voice over IP.

The transport layer offers greater flexibility—and potentially improved performance—over the higher layers, but it requires that you handle such tasks as authentication and encryption yourself.

The good news with the TCP and UDP classes is that you don't have to worry about setting `Proxy` to null. The bad news is that if your only access to the Internet is through a web proxy, you can forget about working directly at the TCP or UDP layer!

With TCP, you have a choice of either the easier-to-use `TcpClient` and `TcpListener` façade classes, or the feature-rich `Socket` class. (In fact, you can mix and match, because `TcpClient` exposes the underlying `Socket` object through the `Client` property.) The `Socket` class exposes more configuration options and allows direct access to the network layer (IP) and non-Internet-based protocols such as Novell's SPX/IPX.

As with other protocols, TCP differentiates a client and server: the client initiates a

As with other protocols, TCP differentiates a client and server: the client initiates a request, while the server waits for a request. Here's the basic structure for a TCP client request:

```
using (TcpClient client = new TcpClient ("address", port))
using (NetworkStream n = client.GetStream())
{
  // Read and write to the network stream...
}
```

`TcpClient` immediately establishes a connection upon construction to a server at the given IP or domain name address and port. The constructor blocks until a connection is established. The `NetworkStream` then provides a means of two-way communication, for both transmitting and receiving bytes of data from a server.

A simple TCP server looks like this:

```
TcpListener listener = new TcpListener (<ip address>, port);
listener.Start();
```

```
while (keepProcessingRequests)
  using (TcpClient c = listener.AcceptTcpClient())
```

```
  using (TcpClient c = listener.AcceptTcpClient())
  using (NetworkStream n = c.GetStream())
  {
    // Read and write to the network stream...
  }
listener.Stop();
```


`TcpListener` requires the local IP address on which to listen (a computer with two network cards, for instance, may have two addresses). You can use `IPAddress.Any` to tell it to listen on all (or the only) local IP addresses. `AcceptTcpClient` blocks until a client request is received, at which point we call `GetStream`, just as on the client side.

When working at the transport layer, you need to decide on a protocol for who talks when, and for how long—rather like with a walkie-talkie. If both parties talk or listen at the same time, communication breaks down!

Let's invent a protocol where the client speaks first, saying "Hello," and then the

Let's invent a protocol where the client speaks first, saying "Hello," and then the server responds by saying "Hello right back!" Here's the code:

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Threading;

class TcpDemo
{
  static void Main()
  {
```

```
    {
      new Thread (Server).Start();
      Thread.Sleep (500);
      Client();
    }

  // Run server method concurrently.
  // Give server time to start.

static void Client()
{
  using (TcpClient client = new TcpClient ("localhost", 51111))
  using (NetworkStream n = client.GetStream())
  {
    BinaryWriter w = new BinaryWriter (n);
    w.Write ("Hello");
```

```
    w.Write ("Hello");
    w.Flush();
    Console.WriteLine (new BinaryReader (n).ReadString());
  }
}
```

```
static void Server()     // Handles a single client request, then exits.
{
  TcpListener listener = new TcpListener (IPAddress.Any, 51111);
  listener.Start();
  using (TcpClient c = listener.AcceptTcpClient())
  using (NetworkStream n = c.GetStream())
  {
    string msg = new BinaryReader (n).ReadString();
    BinaryWriter w = new BinaryWriter (n);
    w.Write (msg + " right back!");
    w.Flush();                          // Must call Flush because we're not
  }                                     // disposing the writer.
  listener.Stop();
}
}
```


```
Hello
Hello right back!
```

In this example, we're using the `localhost` loopback to run the client and server on

In this example, we're using the `localhost` loopback to run the client and server on the same machine. We've arbitrarily chosen a port in the unallocated range (above 49152) and used a `BinaryWriter` and `BinaryReader` to encode the text messages. We've avoided closing or disposing the readers and writers in order to keep the underlying `NetworkStream` open until our conversation completes.

`BinaryReader` and `BinaryWriter` might seem like odd choices for reading and writing strings. However, they have a major advantage over `StreamReader` and `StreamWriter`: they prefix strings with an integer indicating the length, so a `BinaryReader` always knows exactly how many bytes to read. If you call `StreamReader.ReadToEnd` you might block indefinitely—because a `NetworkStream` doesn't have an end! As long as the connection is open, the network stream can never be sure that the client isn't going to send more data.

`StreamReader` is in fact completely out of bounds with `NetworkStream`, even if you plan only to call `ReadLine`. This is because `StreamReader` has a read-ahead buffer, which can result in it reading more bytes than are currently available, blocking indefinitely (or until the socket times out). Other streams such as `FileStream` don't suffer this incompatibility with `StreamReader` because they have a definite *end*—at which point

as `FileStream` don't suffer this incompatibility with `StreamReader` because they have a definite *end*—at which point `Read` returns immediately with a value of `0`.

Concurrency

You'll often want to do other things at the same time as reading or writing a TCP stream. If you need to manage just a few concurrent activities, any multithreading option described in Chapters 21 and 22 is viable: a new thread, a TPL `Task`, asynchronous delegates, `ThreadPool.QueueUserWorkItem` or `BackgroundWorker`. On a highly concurrent server, however, you need to be choosier. As a simple rule of thumb:

-
-

For less than 50 concurrent connections, think *simplicity* and use the Task Parallel Library or `ThreadPool.QueueUserWorkItem`.

For more than 50 concurrent connections, think *efficiency* and go for asynchronous methods.

chronous methods.

Chapter 23 describes how to write a TCP server using each of these models.

Receiving POP3 Mail with TCP

The .NET Framework provides no application-layer support for POP3, so you have to write at the TCP layer in order to receive mail from a POP3 server. Fortunately, this is a simple protocol; a POP3 conversation goes like this:

Client	Mail server	Notes
Client connects...	`+OK Hello there.`	Welcome message

```
USER joe
PASS password
```

```
PASS password
LIST
RETR 1
DELE 1
```

```
+OK Password required.
+OK Logged in.
+OK
1 1876
2 5412
3 845
.
+OK 1876 octets
```

Content of message #1

Content of message #1...

.

+OK Deleted.

Lists the ID and file size of each message on the server

Retrieves the message with the specified ID

Deletes a message from the server

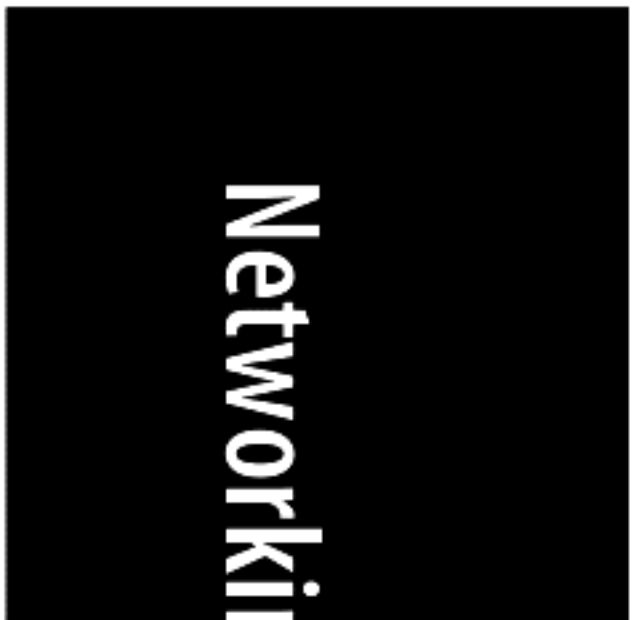
QUIT	+OK Bye-bye.

Each command and response is terminated by a new line (CR + LF) except for the multiline LIST and RETR commands, which are terminated by a single dot on a separate line. Because we can't use `StreamReader` with `NetworkStream`, we can start by writing a helper method to read a line of text in a nonbuffered fashion:

```
static string ReadLine (Stream s)
{
  List<byte> lineBuffer = new List<byte>();
```

```
static string ReadLine (Stream s)
{
  List<byte> lineBuffer = new List<byte>();
  while (true)
  {
    int b = s.ReadByte();
    if (b == 10 || b < 0) break;
    if (b != 13) lineBuffer.Add ((byte)b);
  }
  return Encoding.UTF8.GetString (lineBuffer.ToArray());
}
```

We also need a helper method to send a command. Because we always expect to receive a response starting with "+OK," we can read and validate the response at the same time:




```
static void SendCommand (Stream stream, string line)
{
  byte[] data = Encoding.UTF8.GetBytes (line + "\r\n");
  stream.Write (data, 0, data.Length);
  string response = ReadLine (stream);
  if (!response.StartsWith ("+OK"))
    throw new Exception ("POP Error: " + response);
}
```

With these methods written, the job of retrieving mail is easy. We establish a TCP connection on port 110 (the default POP3 port), and then start talking to the server.

In this example, we write each mail message to a randomly named file with an *.eml* extension, before deleting the message off the server:

```
using (TcpClient client = new TcpClient ("mail.isp.com", 110))
using (NetworkStream n = client.GetStream())
{
  ReadLine (n);                                  // Read the welcome message.
  SendCommand (n, "USER username");
  SendCommand (n, "PASS password");
  SendCommand (n, "LIST");                       // Retrieve message IDs
  List<int> messageIDs = new List<int>();
  while (true)
  {
    string line = ReadLine (n);                  // e.g.,  "1 1876"
    if (line == ".") break;
    messageIDs.Add (int.Parse (line.Split (' ')[0] ));   // Message ID
  }

foreach (int id in messageIDs)                   // Retrieve each message.
{
  SendCommand (n, "RETR " + id);
  string randomFile = Guid.NewGuid().ToString() + ".eml";
  using (StreamWriter writer = File.CreateText (randomFile))
```

```
    using (StreamWriter writer = File.CreateText (randomFile))
      while (true)
      {
        string line = ReadLine (n);       // Read next line of message.
        if (line == ".") break;           // Single dot = end of message.
        if (line == "..") line = ".";     // "Escape out" double dot.
        writer.WriteLine (line);          // Write to output file.
      }
    SendCommand (n, "DELE " + id);        // Delete message off server.
  }
  SendCommand (n, "QUIT");
}
```


1

6

Serialization

This chapter introduces serialization and deserialization, the mechanism by which objects can be represented in a flat text or binary form. Unless otherwise stated, the types in this chapter all exist in the following namespaces:

objects can be represented in a flat text or binary form. Unless otherwise stated, the types in this chapter all exist in the following namespaces:

`System.Runtime.Serialization`
`System.Xml.Serialization`

Serialization Concepts

Serialization is the act of taking an in-memory object or *object graph* (set of objects that reference each other) and flattening it into a stream of bytes or XML nodes that can be stored or transmitted. *Deserialization* works in reverse, taking a data stream and reconstituting it into an in-memory object or object graph.

Serialization and deserialization are typically used to:

-
-

- Transmit objects across a network or application boundary.
- Store representations of objects within a file or database.

Another, less common use is to deep-clone objects. The data contract and XML serialization engines can also be used as general-purpose tools for loading and saving XML files of a known structure.

The .NET Framework supports serialization and deserialization both from the perspective of clients wanting to serialize and deserialize objects, and from the perspective of types wanting some control over how they are serialized.

Serialization Engines

There are four serialization mechanisms in the .NET Framework:

-

-
-
-
-

The data contract serializer

The binary serializer

The (attribute-based) XML serializer (`XmlSerializer`)

The `IXmlSerializable` interface

Of these, the first three are serialization "engines" that do most or all of the serialization work for you. The last is just a hook for doing the serialization yourself, using `XmlReader` and `XmlWriter`. `IXmlSerializable` can work in conjunction with the data contract serializer or `XmlSerializer`, to handle the more complicated XML serialization tasks.

Table 16-1 compares each of the engines.

Table 16-1. Serialization engine comparison

Feature	Data contract serializer	Binary serializer	XmlSerializer	IXmlSerializable
Level of automation	***	*****	****	*
Type coupling				
Version tolerance				
Choice				
*****				
Tight				
***				
Loose				

Loose

*****

Loose

*****

Preserves object references

Can serialize nonpublic fields

Suitability for interoperable messaging

Choice

Yes

*****

Yes

Yes
Yes
**
No
No
****
Choice
Yes
****

Flexibility in reading/writing XML files	**	-	****	*****
Compact output	**	****	**	**
Performance	***	****	* to ***	***

The scores for `IXmlSerializable` assume you've (hand) coded optimally using

The scores for `IXmlSerializable` assume you've (hand) coded optimally using `XmlReader` and `XmlWriter`. The XML serialization engine requires that you recycle the same `XmlSerializer` object for good performance.

Why three engines?

The reason for three engines is partly historical. The Framework started out with two distinct goals in serialization:

-
-

Serializing .NET object graphs with type and reference fidelity

Interoperating with XML and SOAP messaging standards

The first was led by the requirements of Remoting; the second, by Web Services. The job of writing one serialization engine to do both was too daunting, so Microsoft wrote two engines: the binary serializer and the XML serializer.

When Windows Communication Foundation (WCF) was later written, as part of

When Windows Communication Foundation (WCF) was later written, as part of Framework 3.0, part of the goal was to unify Remoting and Web Services. This required a new serialization engine—hence, the *data contract serializer*. The data contract serializer unifies the features of the older two engines *relevant to (interoperable) messaging*. Outside of this context, however, the two older engines are still important.

The data contract serializer

The data contract serializer is the newest and the most versatile of the three serialization engines and is used by WCF. The serializer is particularly strong in two scenarios:

-
-

When exchanging information through standards-compliant messaging protocols

When you need good version tolerance, plus the option of preserving object references

When you need good version tolerance, plus the option of preserving object references

The data contract serializer supports a *data contract* model that helps you decouple the low-level details of the types you want to serialize from the structure of the serialized data. This provides excellent version tolerance, meaning you can deserialize data that was serialized from an earlier or later version of a type. You can even deserialize types that have been renamed or moved to a different assembly.

The data contract serializer can cope with most object graphs, although it can require more assistance than the binary serializer. It can also be used as a general-purpose tool for reading/writing XML files, if you're flexible on how the XML is structured. (If you need to store data in attributes or cope with XML elements presenting in a random order, you cannot use the data contract serializer.)

The binary serializer

The binary serialization engine is easy to use, highly automatic, and well supported throughout the .NET Framework. Remoting uses binary serialization—including when communicating between two application domains in the same process (see Chapter 24).

Chapter 24).

The binary serializer is highly automated: quite often, a single attribute is all that's required to make a complex type fully serializable. The binary serializer is also faster than the data contract serializer when full type fidelity is needed. However, it tightly couples a type's internal structure to the format of the serialized data, resulting in poor version tolerance. (Prior to Framework 2.0, even adding a simple field was a version-breaking change.) The binary engine is also not really designed to produce XML, although it offers a formatter for SOAP-based messaging that provides limited interoperability with simple types.

XmlSerializer

The XML serialization engine can *only* produce XML, and it is less powerful than other engines in saving and restoring a complex object graph (it cannot restore shared object references). It's the most flexible of the three, however, in following

an arbitrary XML structure. For instance, you can choose whether properties are serialized to elements or attributes and the handling of a collection's outer element. The XML engine also provides excellent version tolerance.

`XmlSerializer` is used by ASMX Web Services.

IXmlSerializable

Implementing `IXmlSerializable` means to do the serialization yourself with an `XmlReader` and `XmlWriter`. The `IXmlSerializable` interface is recognized both by

Implementing `IXmlSerializable` means to do the serialization yourself with an `XmlReader` and `XmlWriter`. The `IXmlSerializable` interface is recognized both by `XmlSerializer` and by the data contract serializer, so it can be used selectively to handle the more complicated types. (It also can be used directly by WCF and ASMX Web Services.) We describe `XmlReader` and `XmlWriter` in detail in Chapter 11.

Formatters

The output of the data contract and binary serializers is shaped by a pluggable *formatter*. The role of a formatter is the same with both serialization engines, although they use completely different classes to do the job.

A formatter shapes the final presentation to suit a particular medium or context of serialization. In general, you can choose between XML and binary formatters. An XML formatter is designed to work within the context of an XML reader/writer, text file/stream, or SOAP messaging packet. A binary formatter is designed to work in a context where an arbitrary stream of bytes will do—typically a file/stream or proprietary messaging packet. Binary output is usually smaller than XML—sometimes radically so.

The term "binary" in the context of a formatter is unrelated to the "binary" serialization engine. Each of the two engines ships with both XML and binary formatters!

In theory, the engines are decoupled from their formatters. In practice, the design of each engine is geared toward one kind of formatter. The data contract serializer is geared toward the interoperability requirements of XML messaging. This is good for the XML formatter but means its binary formatter doesn't always achieve the gains you might hope. In contrast, the binary engine provides a relatively good binary formatter, but its XML formatter is highly limited, offering only crude SOAP interoperability.

Explicit Versus Implicit Serialization

Serialization and deserialization can be initiated in two ways.

The first is *explicitly*, by requesting that a particular object be serialized or deserialized. When you serialize or deserialize explicitly, you choose both the serialization engine and the formatter.

ized. When you serialize or deserialize explicitly, you choose both the serialization engine and the formatter.

In contrast, *implicit* serialization is initiated by the Framework. This happens when:

-
-

A serializer recurses a child object.

You use a feature that relies on serialization, such as WCF, Remoting, or Web Services.

WCF always uses the data contract serializer, although it can interoperate with the attributes and interfaces of the other engines.

Remoting always uses the binary serialization engine.

Web Services always uses `XmlSerializer`.

The Data Contract Serializer

Here are the basic steps in using the data contract serializer:

1. Decide whether to use the `DataContractSerializer` or the `NetDataContract-Serializer`.
2. Decorate the types and members you want to serialize with `[DataContract]` and `[DataMember]` attributes, respectively.
3. Instantiate the serializer and call `WriteObject` or `ReadObject`.

If you chose the `DataContractSerializer`, you will also need to register "known types" (subtypes that can also be serialized), and decide whether to preserve object references.

You may also need to take special action to ensure that collections are properly serialized.

Types for the data contract serializer are defined in the

Types for the data contract serializer are defined in the `System.Runtime.Serialization` namespace, in an assembly of the same name.

DataContractSerializer Versus NetDataContractSerializer

There are two data contract serializers:

`DataContractSerializer`
: Loosely couples .NET types to data contract types

`NetDataContractSerializer`
: Tightly couples .NET types to data contract types

The `DataContractSerializer` can produce interoperable standards-compliant XML such as this:

```
<Person xmlns="...">
  ...
</Person>
```


It requires, however, that you explicitly register serializable subtypes in advance so

It requires, however, that you explicitly register serializable subtypes in advance so that it can map a data contract name such as "Person" to the correct .NET type. The `NetDataContractSerializer` requires no such assistance, because it writes the full type and assembly names of the types it serializes, rather like the binary serialization engine:

```
<Person z:Type="SerialTest.Person" z:Assembly=
  "SerialTest, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
  ...
</Person>
```

Such output, however, is proprietary. It also relies on the presence of a specific.NET type in a specific namespace and assembly in order to deserialize.

If you're saving an object graph to a "black box," you can choose either serializer, depending on what benefits are more important to you. If you're communicating through WCF, or reading/writing an XML file, you'll most likely want the `DataContractSerializer`.

Another difference between the two serializers is that `NetDataContractSerializer` always preserves referential equality; `DataContractSerializer` does so only upon request.

always preserves referential equality, `DataContractSerializer` does so only upon request.

We'll go into each of these topics in more detail in the following sections.

Using the Serializers

After choosing a serializer, the next step is to attach attributes to the types and members you want to serialize. At a minimum:

-
-

Add the `[DataContract]` attribute to each type.

Add the `[DataMember]` attribute to each member that you want to include.

Here's an example:

```
namespace SerialTest
```

```
namespace SerialTest
{
   [DataContract] public class Person
   {
      [DataMember] public string Name;
      [DataMember] public int Age;
   }
}
```

These attributes are enough to make a type *implicitly* serializable through the data contract engine.

You can then *explicitly* serialize or deserialize an object instance by instantiating a `DataContractSerializer` or `NetDataContractSerializer` and calling `WriteObject` or `ReadObject`:

ReadObject:

```
    Person p = new Person { Name = "Stacey", Age = 30 };

var ds = new DataContractSerializer (typeof (Person));
```



```
using (Stream s = File.Create ("person.xml"))
  ds.WriteObject (s, p);
Person p2;
using (Stream s = File.OpenRead ("person.xml"))
  p2 = (Person) ds.ReadObject (s);

// Serialize
// Deserialize

Console.WriteLine (p2.Name + " " + p2.Age);
```

```
Console.WriteLine (p2.Name + " " + p2.Age);

// Stacey 30
```

`DataContractSerializer`'s constructor requires the *root object* type (the type of the object you're explicitly serializing). In contrast, `NetDataContractSerializer` does not:

```
var ns = new NetDataContractSerializer();

// NetDataContractSerializer is otherwise the same to use
// as DataContractSerializer.
...
```

Both types of serializer use the XML formatter by default. With an `XmlWriter`, you can request that the output be indented for readability:

```
Person p = new Person { Name = "Stacey", Age = 30 };
var ds = new DataContractSerializer (typeof (Person));
```

```
var ds = new DataContractSerializer (typeof (Person));

XmlWriterSettings settings = new XmlWriterSettings() { Indent = true };
using (XmlWriter w = XmlWriter.Create ("person.xml", settings))
  ds.WriteObject (w, p);

System.Diagnostics.Process.Start ("person.xml");
```

Here's the result:

```
<Person xmlns="http://schemas.datacontract.org/2004/07/SerialTest"
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Age>30</Age>
  <Name>Stacey</Name>
</Person>
```

The XML element name `<Person>` reflects the *data contract name*, which, by default, is the .NET type name. You can override this and explicitly state a data contract name as follows:

```
[DataContract (Name="Candidate")]
```

```
[DataContract (Name="Candidate")]
public class Person { ... }
```

The XML namespace reflects the *data contract namespace*, which, by default, is *http://schemas.datacontract.org/2004/07/*, plus the .NET type namespace. You can override this in a similar fashion:

```
[DataContract (Namespace="http://oreilly.com/nutshell")]
public class Person { ... }
```


Specifying a name and namespace decouples the contract identity from the .NET type name. It ensures that, should you later refactor and change the type's name or namespace, serialization is unaffected.

You can also override names for data members:

```
[DataContract (Name="Candidate", Namespace="http://oreilly.com/nutshell")]
public class Person
{
  [DataMember (Name="FirstName")]  public string Name;
  [DataMember (Name="ClaimedAge")] public int Age;
}
```

}

Here's the output:

```
<?xml version="1.0" encoding="utf-8"?>
<Candidate xmlns="http://oreilly.com/nutshell"
           xmlns:i="http://www.w3.org/2001/XMLSchema-instance" >
  <ClaimedAge>30</ClaimedAge>
  <FirstName>Stacey</FirstName>
</Candidate>
```

[DataMember] supports both fields and properties—public and private. The field or property's data type can be any of the following:

-
-
-
-
-
-

- Any primitive type
- `DateTime`, `TimeSpan`, `Guid`, `Uri`, or an `Enum` value
- Nullable versions of the above

`byte[]` (serializes in XML to base 64)

Any "known" type decorated with `DataContract`

Any `IEnumerable` type (see the section "Serializing Collections" on page 642, later in this chapter)

Any type with the `[Serializable]` attribute or implementing `ISerializable` (see the section "Extending Data Contracts" on page 625 later in this chapter)

Any type implementing `IXmlSerializable`

Specifying a binary formatter

You can use a binary formatter with `DataContractSerializer` or `NetDataContract`

Specifying a binary formatter

You can use a binary formatter with `DataContractSerializer` or `NetDataContractSerializer`. The process is the same:

```
Person p = new Person { Name = "Stacey", Age = 30 };
var ds = new DataContractSerializer (typeof (Person));

var s = new MemoryStream();
using (XmlDictionaryWriter w = XmlDictionaryWriter.CreateBinaryWriter (s))
  ds.WriteObject (w, p);

var s2 = new MemoryStream (s.ToArray());
Person p2;

using (XmlDictionaryReader r = XmlDictionaryReader.CreateBinaryReader (s2,
                                 XmlDictionaryReaderQuotas.Max))
  p2 = (Person) ds.ReadObject (r);
```

The output varies between being slightly smaller than that of the XML formatter, and radically smaller if your types contain large arrays.

The output varies between being slightly smaller than that of the XML formatter, and radically smaller if your types contain large arrays.

Serializing Subclasses

You don't need to do anything special to handle the serializing of subclasses with the `NetDataContractSerializer`. The only requirement is that subclasses have the `DataContract` attribute. The serializer will write the fully qualified names of the actual types that it serializes as follows:

```
<Person ... z:Type="SerialTest.Person" z:Assembly=
  "SerialTest, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
```

A `DataContractSerializer`, however, must be informed about all subtypes that it may have to serialize or deserialize. To illustrate, suppose we subclass `Person` as follows:

```
[DataContract] public class Person
{
  [DataMember] public string Name;
```

```
  [DataMember] public string Name;
  [DataMember] public int Age;
}
[DataContract] public class Student : Person { }
[DataContract] public class Teacher : Person { }
```

and then write a method to clone a Person:

```
static Person DeepClone (Person p)
{
  var ds = new DataContractSerializer (typeof (Person));
  MemoryStream stream = new MemoryStream();
  ds.WriteObject (stream, p);
  stream.Position = 0;
  return (Person) ds.ReadObject (stream);
}
```

which we call as follows:

```
Person  person  = new Person  { Name = "Stacey", Age = 30 };
Student student = new Student { Name = "Stacey", Age = 30 };
Teacher teacher = new Teacher { Name = "Stacey", Age = 30 };
```

```
Person  p2 =                DeepClone (person);
Student s2 = (Student)      DeepClone (student);
Teacher t2 = (Teacher)      DeepClone (teacher);
```

```
// OK
// SerializationException
// SerializationException
```

`DeepClone` works if called with a `Person` but throws an exception with a `Student` or `Teacher`, because the deserializer has no way of knowing what .NET type (or assembly) a "Student" or "Teacher" should resolve to. This also helps with security, in that it prevents the deserialization of unexpected types.

The solution is to specify all permitted or "known" subtypes. You can do this either when constructing the `DataContractSerializer`:

The solution is to specify all permitted or "known" subtypes. You can do this either when constructing the `DataContractSerializer`:

```
var ds = new DataContractSerializer (typeof (Person),
  new Type[] { typeof (Student), typeof (Teacher) } );
```

or in the type itself, with the `KnownType` attribute:

```
[DataContract, KnownType (typeof (Student)), KnownType (typeof (Teacher))]
public class Person
...
```

Here's what a serialized `Student` now looks like:

```
<Person xmlns="..."
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
        i:type="Student" >
   ...
<Person>
```

Because we specified `Person` as the root type, the root element still has that name. The actual subclass is described separately—in the `type` attribute.

The `NetDataContractSerializer` suffers a performance hit when serializing subtypes—with either formatter. It seems that when it encounters a subtype, it has to stop and think for a while!

Serialization performance matters on an application server that's handling many concurrent requests.

Object References

References to other objects are serialized, too. Consider the following classes:

```
[DataContract] public class Person
{
  [DataMember] public string Name;
  [DataMember] public int Age;
  [DataMember] public Address HomeAddress;
}
```

```
[DataContract] public class Address
```

```
[DataContract] public class Address
{
  [DataMember] public string Street, Postcode;
}
```

Here's the result of serializing this to XML using the `DataContractSerializer`:

```
<Person...>
  <Age>...</Age>
  <HomeAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </HomeAddress>
  <Name>...</Name>
```

```
    <Name>...</Name>
  </Person>
```


The `DeepClone` method we wrote in the preceding section would clone `HomeAddress`, too—distinguishing it from a simple `MemberwiseClone`.

If you're using a `DataContractSerializer`, the same rules apply when subclassing `Address` as when subclassing the root type. So, if we define a `USAddress` class, for instance:

```
[DataContract]
public class USAddress : Address { }
```

and assign an instance of it to a `Person`:

and assign an instance of it to a `Person`:

```
Person p = new Person { Name = "John", Age = 30 };
p.HomeAddress = new USAddress { Street="Fawcett St", Postcode="02138" };
```

`p` could not be serialized. The solution is either to apply the `KnownType` attribute to `Address`, as shown next:

```
[DataContract, KnownType (typeof (USAddress))]
public class Address
{
  [DataMember] public string Street, Postcode;
}
```

or to tell `DataContractSerializer` about `USAddress` in construction:

```
var ds = new DataContractSerializer (typeof (Person),
  new Type[] { typeof (USAddress) } );
```

(We don't need to tell it about `Address` because it's the declared type of the

(We don't need to tell it about `Address` because it's the declared type of the `HomeAddress` data member.)

Preserving object references

The `NetDataContractSerializer` always preserves referential
`DataContractSerializer` does not, unless you specifically ask it to.

equality.

The

This means that if the same object is referenced in two different places, a `DataContractSerializer` ordinarily writes it twice. So, if we modify the preceding example so that `Person` also stores a work address:

```
[DataContract] public class Person
{
  ...
```

```
  {
    ...
    [DataMember] public Address HomeAddress, WorkAddress;
  }
```

and then serialize an instance as follows:

```
Person p = new Person { Name = "Stacey", Age = 30 };
p.HomeAddress = new Address { Street = "Odo St", Postcode = "6020" };
```

```
p.HomeAddress = new Address { Street = "Odo St", Postcode = "6020" };
p.WorkAddress = p.HomeAddress;
```


we would see the same address details twice in the XML:

```
...
<HomeAddress>
  <Postcode>6020</Postcode>
  <Street>Odo St</Street>
</HomeAddress>
...
<WorkAddress>
  <Postcode>6020</Postcode>
  <Street>Odo St</Street>
```

```
    <Street>Odo St</Street>
  </WorkAddress>
```

When this was later deserialized, `WorkAddress` and `HomeAddress` would be different objects. The advantage of this system is that it keeps the XML simple and standards-compliant. The disadvantages of this system include larger XML, loss of referential integrity, and the inability to cope with cyclical references.

You can request referential integrity by specifying `true` for `preserveObjectReferences` when constructing a `DataContractSerializer`:

```
var ds = new DataContractSerializer (typeof (Person),
                                     null, 1000, false, true, null);
```

The third argument is mandatory when `preserveObjectReferences` is `true`: it indicates the maximum number of object references that the serializer should keep track of. The serializer throws an exception if this number is exceeded (this prevents a denial of service attack through a maliciously constructed stream).

Here's what the XML then looks like for a `Person` with the same home and work addresses:

addresses:

```
<Person xmlns="http://schemas.datacontract.org/2004/07/SerialTest"
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"
        z:Id="1">
  <Age>30</Age>
  <HomeAddress z:Id="2">
    <Postcode z:Id="3">6020</Postcode>
    <Street z:Id="4">Odo St</Street>
  </HomeAddress>
  <Name z:Id="5">Stacey</Name>
  <WorkAddress z:Ref="2" i:nil="true" />
</Person>
```

The cost of this is in reduced interoperability (notice the proprietary namespace of the `Id` and `Ref` attributes).

Version Tolerance

You can add and remove data members without breaking forward or backward

You can add and remove data members without breaking forward or backward compatibility. By default, the data contract deserializers do the following:

-
-

Skip over data for which there is no `[DataMember]` in the type.

Don't complain if any `[DataMember]` is missing in the serialization stream.

Rather than skipping over unrecognized data, you can instruct the deserializer to store unrecognized data members in a black box, and then replay them should the type later be reserialized. This allows you to correctly round-trip data that's been serialized by a later version of your type. To activate this feature, implement `IExtensibleDataObject`. This interface really means "IBlackBoxProvider." It requires that you implement a single property, to get/set the black box:

```
[DataContract] public class Person : IExtensibleDataObject{
  [DataMember] public string Name;
  [DataMember] public int Age;
```

```
  ExtensionDataObject IExtensibleDataObject.ExtensionData { get; set; }
}
```

Required members

If a member is essential for a type, you can demand that it be present with `IsRequired`:

```
[DataMember (IsRequired=true)] public int ID;
```

If that member is not present, an exception is then thrown upon deserialization.

Member Ordering

The data contract serializers are extremely fussy about the ordering of data members. The deserializers, in fact, *skip over any members considered out of sequence*.

Members are written in the following order when serializing:

1. Base class to subclass

1. Base class to subclass

2. Low `Order` to high `Order` (for data members whose `Order` is set)
3. Alphabetical order (using *ordinal* string comparison)

So, in the preceding examples, `Age` comes before `Name`. In the following example, `Name` comes before `Age`:

```
[DataContract] public class Person
{
  [DataMember (Order=0)] public string Name;
  [DataMember (Order=1)] public int Age;
}
```

If `Person` has a base class, the base class's data members would all serialize first.

The main reason to specify an order is to comply with a particular XML schema. XML element order equates to data member order.

If you don't need to interoperate with anything else, the easiest approach is *not* to specify a member `Order` and rely purely on alphabetical ordering. A discrepancy will then never arise between serialization and deserialization as members are added and removed. The only time you'll come unstuck is if you move a member between a base class and a subclass.

Null and Empty Values

Null and Empty Values

There are two ways to deal with a data member whose value is null or empty:

1. Explicitly write the null or empty value (the default).
2. Omit the data member from the serialization output.

In XML, an explicit null value looks like this:

```
<Person xmlns="..."
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Name i:nil="true" />
</Person>
```

Writing null or empty members can waste space, particularly on a type with lots of fields or properties that are usually left empty. More importantly, you may need to follow an XML schema that expects the use of optional elements (e.g., `minOccurs="0"`) rather than `nil` values.

minOccurs="0") rather than nil values.

You can instruct the serializer not to emit data members for null/empty values as follows:

```
[DataContract] public class Person
{
  [DataMember (EmitDefaultValue=false)] public string Name;
  [DataMember (EmitDefaultValue=false)] public int Age;
}
```

Name is omitted if its value is null; Age is omitted if its value is 0 (the default value for the int type).

The data contract deserializer, in rehydrating an object, bypasses the type's constructors and field initializers. This allows you to omit data members as described without breaking fields that are assigned nondefault values through an initializer or constructor. To illustrate, suppose we set the default Age for a Person to 30 as follows:

Person to 30 as follows:

```
[DataMember (EmitDefaultValue=false)]
public int Age = 30;
```

Now suppose that we instantiate Person, explicitly set its Age from 30 to 0, and then serialize it. The output won't include Age, because 0 is the default value for the int type. This means that in deserialization, Age will be ignored and the field will remain at its default value—which fortunately is 0, given that field initializers and constructors were bypassed.

Data Contracts and Collections

The data contract serializers can save and repopulate any enumerable collection. For instance, suppose we define Person to have a List<> of addresses:

```
[DataContract] public class Person
{
  ...
  [DataMember] public List<Address> Addresses;
}
[DataContract] public class Address
{
  [DataMember] public string Street, Postcode;
}
```

Here's the result of serializing a `Person` with two addresses:

```
<Person ...>
  ...
```

```
...
<Addresses>
  <Address>
    <Postcode>6020</Postcode>
    <Street>Odo St</Street>
  </Address>
  <Address>
    <Postcode>6152</Postcode>
    <Street>Comer St</Street>
  </Address>
</Addresses>
...
```

```
    ...
</Person>
```

Notice that the serializer doesn't encode any information about the particular *type* of collection it serialized. If the `Addresses` field was instead of type `Address[]`, the output would be identical. This allows the collection type to change between serialization and deserialization without causing an error.

Sometimes, though, you need your collection to be of a more specific type than you expose. An extreme example is with interfaces:

```
[DataMember] public IList<Address> Addresses;
```

This serializes correctly (as before), but a problem arises in deserialization. There's no way the deserializer can know which concrete type to instantiate, so it chooses the simplest option—an array. The deserializer sticks to this strategy even if you initialize the field with a different concrete type:

```
[DataMember] public IList<Address> Addresses = new List<Address>();
```

(Remember that the deserializer bypasses field initializers.) The workaround is to make the data member a private field and add a public property to access it:

make the data member a private field and add a public property to access it:

```
[DataMember (Name="Addresses")] List<Address> _addresses;
```


```
public IList<Address> Addresses { get { return _addresses; } }
```

In a nontrivial application, you would probably use properties in this manner anyway. The only unusual thing here is that we've marked the private field as the data member, rather than the public property.

way. The only unusual thing here is that we've marked the private field as the data member, rather than the public property.

Subclassed Collection Elements

The serializer handles subclassed collection elements transparently. You must declare the valid subtypes just as you would if they were used anywhere else:

```
[DataContract, KnownType (typeof (USAddress))]
public class Address
{
  [DataMember] public string Street, Postcode;
}
```

```
public class USAddress : Address { }
```

Adding a `USAddress` to a `Person`'s address list then generates XML like this:

```
...
  <Addresses>
```

```
...
  <Addresses>
    <Address i:type="USAddress">
      <Postcode>02138</Postcode>
      <Street>Fawcett St</Street>
    </Address>
  </Addresses>
```

Customizing Collection and Element Names

If you subclass a collection class itself, you can customize the XML name used to describe each element by attaching a `CollectionDataContract` attribute:

```
[CollectionDataContract (ItemName="Residence")]
public class AddressList : Collection<Address> { }
```

```
[DataContract] public class Person
{
  ...
  [DataMember] public AddressList Addresses;
```

```
  [DataMember] public AddressList Addresses;
}
```

Here's the result:

```
...
  <Addresses>
    <Residence>
      <Postcode>6020</Postcode
      <Street>Odo St</Street>
    </Residence>
    ...
```

`CollectionDataContract` also lets you specify a `Namespace` and `Name`. The latter is not used when the collection is serialized as a property of another object (such as in this example), but it is when the collection is serialized as the root object.

used when the collection is serialized as a property of another object (such as in this example), but it is when the collection is serialized as the root object.

You can also use `CollectionDataContract` to control the serialization of dictionaries:

```
[CollectionDataContract (ItemName="Entry",
                         KeyName="Kind",
                       ValueName="Number")]
public class PhoneNumberList : Dictionary <string, string> { }
```



```
[DataContract] public class Person
{
  ...
  [DataMember] public PhoneNumberList PhoneNumbers;
}
```

Here's how this formats:

Here's how this formats:

```
...
  <PhoneNumbers>
    <Entry>
      <Kind>Home</Kind>
      <Number>08 1234 5678</Number>
    </Entry>
    <Entry>
      <Kind>Mobile</Kind>
      <Number>040 8765 4321</Number>
    </Entry>
  </PhoneNumbers>
```

Extending Data Contracts

Extending Data Contracts

This section describes how you can extend the capabilities of the data contract serializer through serialization hooks, `[Serializable]` and `IXmlSerializable`.

Serialization and Deserialization Hooks

You can request that a custom method be executed before or after serialization, by flagging the method with one of the following attributes:

`[OnSerializing]`
: Indicates a method to be called just *before* serialization

`[OnSerialized]`
: Indicates a method to be called just *after* serialization

Similar attributes are supported for deserialization:

Similar attributes are supported for deserialization:

`[OnDeserializing]`
: Indicates a method to be called just *before* deserialization

`[OnDeserialized]`
: Indicates a method to be called just *after* deserialization

The custom method must have a single parameter of type `StreamingContext`. This parameter is required for consistency with the binary engine, and it is not used by the data contract serializer.

[OnSerializing] and [OnDeserialized] are useful in handling members that are outside the capabilities of the data contract engine, such as a collection that has an extra payload or that does not implement standard interfaces. Here's the basic approach:

```
[DataContract] public class Person
{
  public SerializationUnfriendlyType Addresses;
```



```
[DataMember (Name="Addresses")]
SerializationFriendlyType _serializationFriendlyAddresses;

[OnSerializing]
void PrepareForSerialization (StreamingContext sc)
{
```

```
  {
    // Copy Addresses-> _serializationFriendlyAddresses
    // ...
  }

  [OnDeserialized]
  void CompleteDeserialization (StreamingContext sc)
  {
    // Copy _serializationFriendlyAddresses-> Addresses
    // ...
  }
}
```

An [OnSerializing] method can also be used to conditionally serialize fields:

```
public DateTime DateOfBirth;

[DataMember] public bool Confidential;
```

```
[DataMember] public bool Confidential;
```

```
[DataMember (Name="DateOfBirth", EmitDefaultValue=false)]
DateTime? _tempDateOfBirth;
```

```
[OnSerializing]
void PrepareForSerialization (StreamingContext sc)
{
  if (Confidential)
    _tempDateOfBirth = DateOfBirth;
  else
    _tempDateOfBirth = null;
}
```

Recall that the data contract deserializers bypass field initializers and constructors. An `[OnDeserializing]` method acts as a pseudoconstructor for deserialization, and it is useful for initializing fields excluded from serialization:

```
[DataContract] public class Test
{
```

```
[DataContract] public class Test
{
  bool _editable = true;
```

```
public Test() { _editable = true; }
```

```
  [OnDeserializing]
  void Init (StreamingContext sc)
  {
    _editable = true;
  }
}
```

If it wasn't for the `Init` method, `_editable` would be false in a deserialized instance of `Test`—despite the other two attempts at making it true.

of Test—despite the other two attempts at making it true.

Methods decorated with these four attributes can be private. If subtypes need to participate, they can define their own methods with the same attributes, and they will get executed, too.

Interoperating with [Serializable]

The data contract serializer can also serialize types marked with the binary serialization engine's attributes and interfaces. This ability is important, since support for the binary engine has been woven into much of what was written prior to Framework 3.0—including the .NET Framework itself!

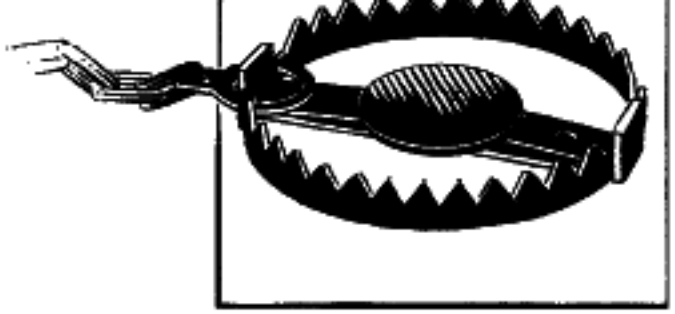
The following things flag a type as being serializable for the binary engine:

- The [Serializable] attribute
- Implementing ISerializable

- The `[Serializable]` attribute
- Implementing `ISerializable`

Binary interoperability is useful in serializing existing types as well as new types that need to support both engines. It also provides another means of extending the capability of the data contract serializer, because the binary engine's `ISerializable` is more flexible than the data contract attributes. Unfortunately, the data contract serializer is inefficient in how it formats data added via `ISerializable`.

A type wanting the best of both worlds cannot define attributes for both engines. This creates a problem for types such as `string` and `DateTime`, which for historical reasons cannot divorce the binary engine attributes. The data contract serializer works around this by filtering out these basic types and processing them specially. For all other types marked for binary serialization, the data contract serializer applies similar rules to what the binary engine would use. This means it honors attributes such as `NonSerialized` or calls `ISerializable` if implemented. It does not *thunk* to the binary engine itself—this ensures that output is formatted in the same style as if data contract attributes were used.



Types designed to be serialized with the binary engine expect object references to be preserved. You can enable this option through the `DataContractSerializer` (or by using the `NetDataContractSerializer`).

The rules for registering known types also apply to objects and subobjects serialized through the binary interfaces.

The following example illustrates a class with a [Serializable] data member:

```
[DataContract] public class Person
{
  ...
  [DataMember] public Address MailingAddress;
}
[Serializable] public class Address
{
  public string Postcode, Street;
}
```

Here's the result of serializing it:

Here's the result of serializing it:

```
<Person ...>
  ...
  <MailingAddress>
    <Postcode>6020</Postcode>
    <Street>Odo St</Street>
  </MailingAddress>
  ...
```

Had `Address` implemented `ISerializable`, the result would be less efficiently formatted:

```
<MailingAddress>
  <Street xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
    i:type="d3p1:string" xmlns="">str</Street>
  <Postcode xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
```

```
  <Postcode xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
    i:type="d3p1:string" xmlns="">pcode</Postcode>
</MailingAddress>
```

Interoperating with IXmlSerializable

A limitation of the data contract serializer is that it gives you little control over the structure of the XML. In a WCF application, this can actually be beneficial, in that it makes it easier for the infrastructure to comply with standard messaging protocols.

If you do need precise control over the XML, you can implement `IXmlSerializable` and then use `XmlReader` and `XmlWriter` to manually read and write the XML. The data contract serializer allows you to do this just on the types for which this level of control is required. We describe the `IXmlSerializable` interface further in the final section of this chapter.

The Binary Serializer

The binary serialization engine is used implicitly by Remoting. It can also be used to perform such tasks as saving and restoring objects to disk. The binary serialization is highly automated and can handle complex object graphs with minimum intervention.

There are two ways to make a type support binary serialization. The first is attribute-based; the second involves implementing `ISerializable`. Adding attributes is

simpler; implementing `ISerializable` is more flexible. You typically implement `ISerializable` to:

-
-

Dynamically control what gets serialized.

Make your serializable type friendly to being subclassed by other parties.

Getting Started

Getting Started

A type can be made serializable with a single attribute:

```
[Serializable] public sealed class Person
{
  public string Name;
  public int Age;
}
```

The [Serializable] attribute instructs the serializer to include all fields in the type. This includes both private and public fields (but not properties). Every field must itself be serializable; otherwise, an exception is thrown. Primitive .NET types such as string and int support serialization (as do many other .NET types).

The Serializable attribute is not inherited, so subclasses are not automatically serializable, unless also marked with this attribute.

With automatic properties, the binary serialization engine seri-

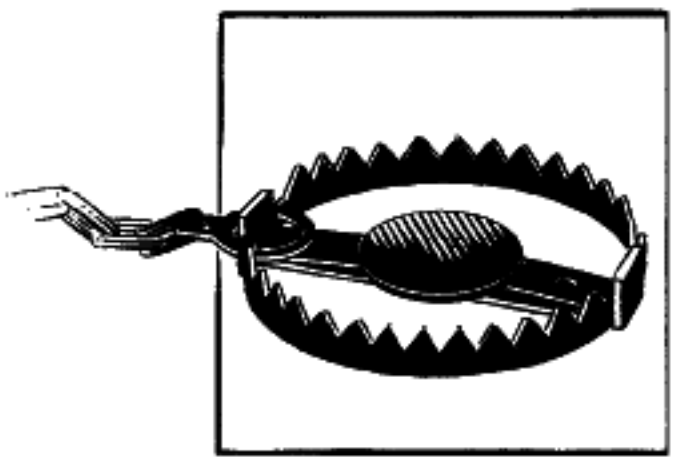
With automatic properties, the binary serialization engine serializes the underlying compiler-generated field. The name of this field, unfortunately, can change when its type is recompiled with more properties, breaking compatibility with existing serialized data. The workaround is either to avoid automatic properties in `[Serializable]` types or to implement `ISerializable`.

To serialize an instance of `Person`, you instantiate a formatter and call `Serialize`. There are two formatters for use with the binary engine:

`BinaryFormatter`
: This is the more efficient of the two, producing smaller output in less time. Its namespace is `System.Runtime.Serialization.Formatters.Binary`.

`SoapFormatter`
: This supports basic SOAP-style messaging when used with Remoting. Its namespace is `System.Runtime.Serialization.Formatters.Soap`.

`BinaryFormatter` is contained in *mscorlib*; `SoapFormatter` is contained in *System.Runtime.Serialization.Formatters.Soap.dll*.



The SoapFormatter is less functional than the BinaryFormatter. The SoapFormatter doesn't support generic types or the filtering of extraneous data necessary for version tolerant serialization.

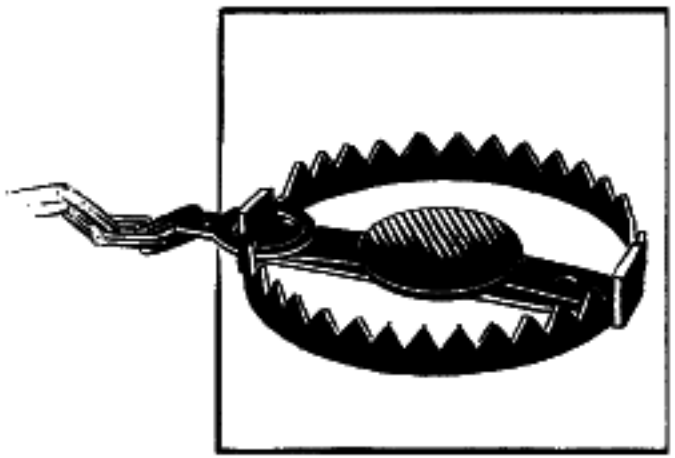
The two formatters are otherwise exactly the same to use. The following serializes a Person with a BinaryFormatter:

```
Person p = new Person() { Name = "George", Age = 25 };

IFormatter formatter = new BinaryFormatter();

using (FileStream s = File.Create ("serialized.bin"))
  formatter.Serialize (s, p);
```

All the data necessary to reconstruct the Person object is written to the file *serialized.bin*. The Deserialize method restores the object:

```
using (FileStream s = File.OpenRead ("serialized.bin"))
```

```
using (FileStream s = File.OpenRead ("serialized.bin"))
{
  Person p2 = (Person) formatter.Deserialize (s);
  Console.WriteLine (p2.Name + " " + p.Age);     // George 25
}
```



The deserializer bypasses all constructors when re-creating objects. Behind the scenes, it calls `FormatterServices.GetUninitializedObject` to do this job. You can call this method yourself to implement some very grubby design patterns!

The serialized data includes full type and assembly information, so if we try to cast the result of deserialization to a matching `Person` type in a different assembly, an error would result. The deserializer fully restores object references to their original

the result of deserialization to a matching `Person` type in a different assembly, an error would result. The deserializer fully restores object references to their original state upon deserialization. This includes collections, which are just treated as serializable objects like any other (all collection types in `System.Collections.*` are marked as serializable).

The binary engine can handle large, complex object graphs without special assistance (other than ensuring that all participating members are serializable). One thing to be wary of is that the serializer's performance degrades in proportion to the number of references in your object graph. This can become an issue in a Remoting server that has to process many concurrent requests.

Binary Serialization Attributes

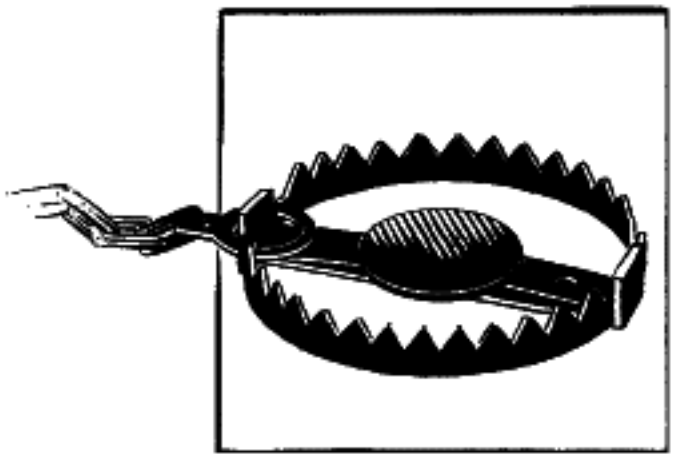
[NonSerialized]

Unlike data contracts, which have an *opt-in* policy in serializing fields, the binary

Unlike data contracts, which have an *opt-in* policy in serializing fields, the binary engine has an *opt-out* policy. Fields that you don't want serialized, such as those used for temporary calculations, or for storing file or window handles, you must mark explicitly with the `[NonSerialized]` attribute:


```
[Serializable] public sealed class Person
{
  public string Name;
  public DateTime DateOfBirth;

  // Age can be calculated, so there's no need to serialize it.
  [NonSerialized] public int Age;
}
```

This instructs the serializer to ignore the `Age` member.



Nonserialized members are always empty or `null` when deserialized—even if field initializers or constructors set them otherwise.

[OnDeserializing] and [OnDeserialized]

Deserialization bypasses all your normal constructors, as well as field initializers. This is of little consequence if every field partakes in serialization, but it can be problematic if some fields are excluded via `[NonSerialized]`. We can illustrate this by adding a `bool` field called `Valid`:

```
public sealed class Person
```

```
public sealed class Person
{
  public string Name;
  public DateTime DateOfBirth;

[NonSerialized] public int Age;
[NonSerialized] public bool Valid = true;

  public Person() { Valid = true; }
}
```

A deserialized `Person` will not be `Valid`—despite the constructor and field initializer.

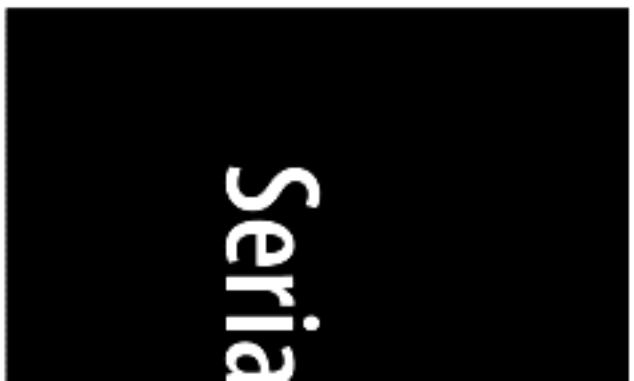
The solution is the same as with the data contract serializer: to define a special deserialization "constructor" with the `[OnDeserializing]` attribute. A method that

The solution is the same as with the data contract serializer: to define a special deserialization "constructor" with the [OnDeserializing] attribute. A method that you flag with this attribute gets called just prior to deserialization:

```
[OnDeserializing]
void OnDeserializing (StreamingContext context)
{
  Valid = true;
}
```

We could also write an [OnDeserialized] method to update the calculated Age field (this fires just *after* deserialization):

```
[OnDeserialized]
void OnDeserialized (StreamingContext context)
{
  TimeSpan ts = DateTime.Now - DateOfBirth;
  Age = ts.Days / 365;                          // Rough age in years
}
```



[OnSerializing] and [OnSerialized]

The binary engine also supports the [OnSerializing] and [OnSerialized] attributes. These flag a method for execution before or after serialization. To see how they can be useful, we'll define a Team class that contains a generic List of players:

```
[Serializable] public sealed class Team
{
  public string Name;
  public List<Person> Players = new List<Person>();
```

```
  public string Name;
  public List<Person> Players = new List<Person>();
}
```

This class serializes and deserializes correctly with the binary formatter but not the SOAP formatter. This is because of an obscure limitation: the SOAP formatter refuses to serialize generic types! An easy solution is to convert `Players` to an array just prior to serialization, then convert it back to a generic `List` upon deserialization. To make this work, we can add another field for storing the array, mark the original `Players` field as `[NonSerialized]`, and then write the conversion code in as follows:

```
[Serializable] public sealed class Team
{
  public string Name;
  Person[] _playersToSerialize;

[NonSerialized] public List<Person> Players = new List<Person>();

[OnSerializing]
void OnSerializing (StreamingContext context)
{
  _playersToSerialize = Players.ToArray();
```