CHAPTER 24

Runtime Serialization

In this chapter:

Serialization is the process of converting an object or a graph of connected objects into a stream of bytes. *Deserialization* is the process of converting a stream of bytes back into its graph of connected objects. The ability to convert objects to and from a byte stream is an incredibly useful mechanism. Here are some examples:

- An application's state (object graph) can easily be saved in a disk file or database and then restored the next time the application is run. ASP.NET saves and restores session state by way of serialization and deserialization.
- A set of objects can easily be copied to the system's clipboard and then pasted into the same or another application. In fact, Windows Forms and Windows Presentation Foundation (WPF) use this.
- A set of objects can be cloned and set aside as a "backup" while a user manipulates the "main" set of objects.
- A set of objects can easily be sent over the network to a process running on another machine. The Microsoft .NET Framework's remoting architecture serializes and deserializes objects that are marshaled by value. It is also used to send objects across AppDomain boundaries, as discussed in Chapter 22, "CLR Hosting and AppDomains."

In addition to the preceding examples, after you have serialized objects in a byte stream in memory, it is quite easy to process the data in more useful ways, such as encrypting and compressing the data.

With serialization being so useful, it is no wonder that many programmers have spent countless hours developing code to perform these types of actions. Historically, this code is difficult to write and is extremely tedious and error-prone. Some of the difficult issues that developers need to grapple with are communication protocols, client/server data type mismatches (such as little-endian/big-endian issues), error handling, objects that refer to other objects, in and out parameters, arrays of structures, and the list goes on.

Well, you'll be happy to know that the .NET Framework has fantastic support for serialization and deserialization built right into it. This means that all of the difficult issues just mentioned are now handled completely and transparently by the .NET Framework. As a developer, you can work with your objects before serialization and after deserialization and have the .NET Framework handle the stuff in the middle.

In this chapter, I explain how the .NET Framework exposes its serialization and deserialization services. For almost all data types, the default behavior of these services will be sufficient, meaning that it is almost no work for you to make your own types serializable. However, there is a small minority of types where the serialization service's default behavior will not be sufficient. Fortunately, the serialization services are very extensible, and I will also explain how to tap into these extensibility mechanisms, allowing you to do some pretty powerful things when serializing or deserializing objects. For example, I'll demonstrate how to serialize Version 1 of an object out to a disk file and then deserialize it a year later into an object of Version 2.

Note This chapter focuses on the runtime serialization technology in the common language runtime (CLR), which has a deep understanding of CLR data types and can serialize all the public, protected, internal, and even private fields of an object to a compressed binary stream for high performance. See the `System.Runtime.Serialization.NetDataContractSerializer` class if you want to serialize CLR data types to an XML stream. The .NET Framework also offers other serialization technologies that are designed more for interoperating between CLR data types and non-CLR data types. These other serialization technologies use the `System.Xml.Serialization.XmlSerializer` class and the `System.Runtime.Serialization.DataContractSerializer` class.

Serialization/Deserialization Quick Start

Let's start off by looking at some code.

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

internal static class QuickStart {
   public static void Main() {
      // Create a graph of objects to serialize them to the stream
      var objectGraph = new List<String> { "Jeff", "Kristin", "Aidan", "Grant" };
      Stream stream = SerializeToMemory(objectGraph);

      // Reset everything for this demo
      stream.Position = 0;
      objectGraph = null;

      // Deserialize the objects and prove it worked
      objectGraph = (List<String>) DeserializeFromMemory(stream);
      foreach (var s in objectGraph) Console.WriteLine(s);
   }

   private static MemoryStream SerializeToMemory(Object objectGraph) {
      // Construct a stream that is to hold the serialized objects
      MemoryStream stream = new MemoryStream();

      // Construct a serialization formatter that does all the hard work
      BinaryFormatter formatter = new BinaryFormatter();

      // Tell the formatter to serialize the objects into the stream
      formatter.Serialize(stream, objectGraph);

      // Return the stream of serialized objects back to the caller
      return stream;
   }

   private static Object DeserializeFromMemory(Stream stream) {
      // Construct a serialization formatter that does all the hard work
      BinaryFormatter formatter = new BinaryFormatter();

      // Tell the formatter to deserialize the objects from the stream
      return formatter.Deserialize(stream);
   }
}
```

Wow, look how simple this is! The `SerializeToMemory` method constructs a `System.IO.MemoryStream` object. This object identifies where the serialized block of bytes is to be placed. Then the method constructs a `BinaryFormatter` object (which can be found in the `System.Runtime.Serialization.Formatters.Binary` namespace). A formatter is a type (implementing the `System.Runtime.Serialization.IFormatter` interface) that knows how to serialize and deserialize an object graph. The Framework Class Library (FCL) ships with two formatters: the `BinaryFormatter` (used in this code example) and a `SoapFormatter` (which can be found in the `System.Runtime.Serialization.Formatters.Soap` namespace and is implemented in the `System.Runtime.Serialization.Formatters.Soap.dll` assembly).

Note As of version 3.5 of the .NET Framework, the `SoapFormatter` class is obsolete and should not be used in production code. However, it can still be useful for debugging serialization code because it produces XML text that you can read. To use XML serialization and deserialization in production code, see the `XmlSerializer` and `DataContractSerializer` classes.

To serialize a graph of objects, just call the formatter's `Serialize` method and pass it two things: a reference to a stream object and a reference to the object graph that you want to serialize. The stream object identifies where the serialized bytes should be placed and can be an object of any type derived from the `System.IO.Stream` abstract base class. This means that you can serialize an object graph to a `MemoryStream`, a `FileStream`, a `NetworkStream`, and so on.

The second parameter to `Serialize` is a reference to an object. This object could be anything: an `Int32`, a `String`, a `DateTime`, an `Exception`, a `List<String>`, a `Dictionary<Int32, DateTime>`, and so on. The object referred to by the `objectGraph` parameter may refer to other objects. For example, `objectGraph` may refer to a collection that refers to a set of objects. These objects may also refer to other objects. When the formatter's `Serialize` method is called, all objects in the graph are serialized to the stream.

Formatters know how to serialize the complete object graph by referring to the metadata that describes each object's type. The `Serialize` method uses reflection to see what instance fields are in each object's type as it is serialized. If any of these fields refer to other objects, then the formatter's `Serialize` method knows to serialize these objects, too.

Formatters have very intelligent algorithms. They know to serialize each object in the graph no more than once out to the stream. That is, if two objects in the graph refer to each other, then the formatter detects this, serializes each object just once, and avoids entering into an infinite loop.

In my `SerializeToMemory` method, when the formatter's `Serialize` method returns, the `MemoryStream` is simply returned to the caller. The application uses the contents of this flat byte array any way it wants. For example, it could save it in a file, copy it to the clipboard, send it over a wire, or whatever.

The `DeserializeFromStream` method deserializes a stream back into an object graph. This method is even simpler than serializing an object graph. In this code, a `BinaryFormatter` is constructed and then its `Deserialize` method is called. This method takes the stream as a parameter and returns a reference to the root object within the deserialized object graph.

Internally, the formatter's `Deserialize` method examines the contents of the stream, constructs instances of all the objects that are in the stream, and initializes the fields in all these objects so that they have the same values they had when the object graph was serialized. Typically, you will cast the object reference returned from the `Deserialize` method into the type that your application is expecting.

Note Here's a fun, useful method that uses serialization to make a deep copy, or clone, of an object.

```
private static Object DeepClone(Object original) {
   // Construct a temporary memory stream
   using (MemoryStream stream = new MemoryStream()) {

      // Construct a serialization formatter that does all the hard work
      BinaryFormatter formatter = new BinaryFormatter();

      // This line is explained in this chapter's "Streaming Contexts" section
      formatter.Context = new StreamingContext(StreamingContextStates.Clone);

      // Serialize the object graph into the memory stream
      formatter.Serialize(stream, original);

      // Seek back to the start of the memory stream before deserializing
      stream.Position = 0;

      // Deserialize the graph into a new set of objects and
      // return the root of the graph (deep copy) to the caller
      return formatter.Deserialize(stream);
   }
}
```

At this point, I'd like to add a few notes to our discussion. First, it is up to you to ensure that your code uses the same formatter for both serialization and deserialization. For example, don't write code that serializes an object graph by using the `SoapFormatter` and then deserializes the graph by using the `BinaryFormatter`. If `Deserialize` can't decipher the contents of the stream, then a `System.Runtime.Serialization.SerializationException` exception will be thrown.

The second thing I'd like to point out is that it is possible and also quite useful to serialize multiple object graphs out to a single stream. For example, let's say that we have the following two class definitions.

```
[Serializable] internal sealed class Customer { /* ... */ }
[Serializable] internal sealed class Order    { /* ... */ }
```

And then, in the main class of our application, we define the following static fields.

```
private static List<Customer> s_customers       = new List<Customer>();
private static List<Order>    s_pendingOrders   = new List<Order>();
private static List<Order>    s_processedOrders = new List<Order>();
```

We can now serialize our application's state to a single stream with a method that looks like this.

```
private static void SaveApplicationState(Stream stream) {
   // Construct a serialization formatter that does all the hard work
   BinaryFormatter formatter = new BinaryFormatter();

   // Serialize our application's entire state
   formatter.Serialize(stream, s_customers);
   formatter.Serialize(stream, s_pendingOrders);
   formatter.Serialize(stream, s_processedOrders);
}
```

To reconstruct our application's state, we would deserialize the state with a method that looks like this.

```
private static void RestoreApplicationState(Stream stream) {
   // Construct a serialization formatter that does all the hard work
   BinaryFormatter formatter = new BinaryFormatter();

   // Deserialize our application's entire state (same order as serialized)
   s_customers       = (List<Customer>) formatter.Deserialize(stream);
   s_pendingOrders   = (List<Order>)    formatter.Deserialize(stream);
   s_processedOrders = (List<Order>)    formatter.Deserialize(stream);
}
```

The third and last thing I'd like to point out has to do with assemblies. When serializing an object, the full name of the type and the name of the type's defining assembly are written to the stream. By default, `BinaryFormatter` outputs the assembly's full identity, which includes the assembly's file name (without extension), version number, culture, and public key information. When deserializing an object, the formatter first grabs the assembly identity and ensures that the assembly is loaded into the executing AppDomain by calling `System.Reflection.Assembly`'s `Load` method (discussed in Chapter 23, "Assembly Loading and Reflection").

After an assembly has been loaded, the formatter looks in the assembly for a type matching that of the object being deserialized. If the assembly doesn't contain a matching type, an exception is thrown and no more objects can be deserialized. If a matching type is found, an instance of the type is created and its fields are initialized from the values contained in the stream. If the type's fields don't exactly match the names of the fields as read from the stream, then a `SerializationException` exception is thrown and no more objects can be deserialized. Later in this chapter, I'll discuss some sophisticated mechanisms that allow you to override some of this behavior.

Important Some extensible applications use `Assembly.LoadFrom` to load an assembly and then construct objects from types defined in the loaded assembly. These objects can be serialized to a stream without any trouble. However, when deserializing this stream, the formatter attempts to load the assembly by calling `Assembly`'s `Load` method instead of calling the `LoadFrom` method. In most cases, the CLR will not be able to locate the assembly file, causing a `SerializationException` exception to be thrown. This catches many developers by surprise—because the objects serialized correctly, the developers expect that the objects will deserialize correctly as well.

If your application serializes objects whose types are defined in an assembly that your application loads using `Assembly.LoadFrom`, then I recommend that you implement a method whose signature matches the `System.ResolveEventHandler` delegate and register this method with `System.AppDomain`'s `AssemblyResolve` event just before calling a formatter's `Deserialize` method. (Unregister this method with the event after `Deserialize` returns.) Now, whenever the formatter fails to load an assembly, the CLR calls your `ResolveEventHandler` method. This method is passed the identity of the assembly that failed to load. The method can extract the assembly file name from the assembly's identity and use this name to construct the path where the application knows the assembly file can be found. Then, the method can call `Assembly.LoadFrom` to load the assembly and return the resulting `Assembly` reference back from the `ResolveEventHandler` method.

This section covered the basics of how to serialize and deserialize object graphs. In the remaining sections, we'll look at what you must do in order to define your own serializable types, and we'll also look at various mechanisms that allow you to have greater control over serialization and deserialization.

Making a Type Serializable

When a type is designed, the developer must make the conscious decision as to whether or not to allow instances of the type to be serializable. By default, types are not serializable. For example, the following code does not perform as expected.

```
internal struct Point { public Int32 x, y; }

private static void OptInSerialization() {
   Point pt = new Point { x = 1, y = 2 };
   using (var stream = new MemoryStream()) {
      new BinaryFormatter().Serialize(stream, pt); // throws SerializationException
   }
}
```

If you were to build and run this code in your program, you'd see that the formatter's `Serialize` method throws a `System.Runtime.Serialization.SerializationException` exception. The problem is that the developer of the `Point` type has not explicitly indicated that `Point` objects may be serialized. To solve this problem, the developer must apply the `System.SerializableAttribute` custom attribute to this type as follows. (Note that this attribute is defined in the `System` namespace, not the `System.Runtime.Serialization` namespace.)

```
[Serializable]
internal struct Point { public Int32 x, y; }
```

Now, if we rebuild the application and run it, it does perform as expected and the `Point` objects will be serialized to the stream. When serializing an object graph, the formatter checks that every object's type is serializable. If any object in the graph is not serializable, the formatter's `Serialize` method throws the `SerializationException` exception.

Note When serializing a graph of objects, some of the object's types may be serializable and some of the objects may not be serializable. For performance reasons, formatters do not verify that all of the objects in the graph are serializable before serializing the graph. So, when serializing an object graph, it is entirely possible that some objects may be serialized to the stream before the `SerializationException` is thrown. If this happens, the stream contains corrupt data. If you think you may be serializing an object graph where some objects may not be serializable, your application code should be able to recover gracefully from this situation. One option is to serialize the objects into a `MemoryStream` first. Then, if all objects are successfully serialized, you can copy the bytes in the `MemoryStream` to whatever stream (for example, file, network) you really want the bytes written to.

The `SerializableAttribute` custom attribute may be applied to reference types (`class`), value types (`struct`), enumerated types (`enum`), and delegate types (`delegate`) only. (Note that enumerated and delegate types are always serializable, so there is no need to explicitly apply the `SerializableAttribute` attribute to these types.) In addition, the `SerializableAttribute` attribute is not inherited by derived types. So, given the following two type definitions, a `Person` object can be serialized, but an `Employee` object cannot.

```
[Serializable]
internal class Person { ... }

internal class Employee : Person { ... }
```

To fix this, you would just apply the `SerializableAttribute` attribute to the `Employee` type as well.

```
[Serializable]
internal class Person { ... }

[Serializable]
internal class Employee : Person { ... }
```

Note that this problem was easy to fix. However, the reverse—defining a type derived from a base type that doesn't have the `SerializableAttribute` attribute applied to it—is not easy to fix. But, this is by design; if the base type doesn't allow instances of its type to be serialized, its fields cannot be serialized, because a base object is effectively part of the derived object. This is why `System.Object` has the `SerializableAttribute` attribute applied to it.

Note In general, it is recommended that most types you define be serializable. After all, this grants a lot of flexibility to users of your types. However, you must be aware that serialization reads all of an object's fields regardless of whether the fields are declared as `public`, `protected`, `internal`, or `private`. You might not want to make a type serializable if it contains sensitive or secure data (like passwords) or if the data would have no meaning or value if transferred.

If you find yourself using a type that was not designed for serialization, and you do not have the source code of the type to add serialization support, all is not lost. In the "Overriding the Assembly and/or Type When Deserializing an Object" section later in this chapter, I will explain how you can make any non-serializable type serializable.

Controlling Serialization and Deserialization

When you apply the `SerializableAttribute` custom attribute to a type, all instance fields (`public`, `private`, `protected`, and so on) are serialized.[1] However, a type may define some instance fields that should not be serialized. In general, there are two reasons why you would not want some of a type's instance fields to be serialized:

- The field contains information that would not be valid when deserialized. For example, an object that contains a handle to a Windows kernel object (such as a file, process, thread, mutex, event, semaphore, and so on) would have no meaning when deserialized into another process or machine because Windows' kernel handles are process-relative values.

[1] Do not use C#'s automatically implemented property feature to define properties inside types marked with the `[Serializable]` attribute, because the compiler generates the names of the fields and the generated names can be different each time that you recompile your code, preventing instances of your type from being deserializable.

- The field contains information that is easily calculated. In this case, you select which fields do not need to be serialized, thus improving your application's performance by reducing the amount of data transferred.

The following code uses the `System.NonSerializedAttribute` custom attribute to indicate which fields of the type should not be serialized. (Note that this attribute is also defined in the `System` namespace, not the `System.Runtime.Serialization` namespace.)

```
[Serializable]
internal class Circle {
   private Double m_radius;

   [NonSerialized]
   private Double m_area;

   public Circle(Double radius) {
      m_radius = radius;
      m_area = Math.PI * m_radius * m_radius;
   }

   ...
}
```

In the preceding code, objects of `Circle` may be serialized. However, the formatter will serialize the values in the object's `m_radius` field only. The value in the `m_area` field will not be serialized because it has the `NonSerializedAttribute` attribute applied to it. This attribute can be applied only to a type's fields, and it continues to apply to this field when inherited by another type. Of course, you may apply the `NonSerializedAttribute` attribute to multiple fields within a type.

So, let's say that our code constructs a `Circle` object as follows.

```
Circle c = new Circle(10);
```

Internally, the `m_area` field is set to a value approximate to 314.159. When this object gets serialized, only the value of the `m_radius` field (10) gets written to the stream. This is exactly what we want, but now we have a problem when the stream is deserialized back into a `Circle` object. When deserialized, the `Circle` object will get its `m_radius` field set to 10, but its `m_area` field will be initialized to 0—not 314.159!

The following code demonstrates how to modify the `Circle` type to fix this problem.

```
[Serializable]
internal class Circle {
   private Double m_radius;

   [NonSerialized]
   private Double m_area;
```

```
   public Circle(Double radius) {
      m_radius = radius;
      m_area = Math.PI * m_radius * m_radius;
   }

   [OnDeserialized]
   private void OnDeserialized(StreamingContext context) {
      m_area = Math.PI * m_radius * m_radius;
   }
}
```

I've changed `Circle` so that it now contains a method marked with the `System.Runtime.Serialization.OnDeserializedAttribute` custom attribute.[2] Whenever an instance of a type is deserialized, the formatter checks whether the type defines a method with this attribute on it and then the formatter invokes this method. When this method is called, all the serializable fields will be set correctly, and they may be accessed to perform any additional work that would be necessary to fully deserialize the object.

In the preceding modified version of `Circle`, I made the `OnDeserialized` method simply calculate the area of the circle by using the `m_radius` field and place the result in the `m_area` field. Now, `m_area` will have the desired value of 314.159.

In addition to the `OnDeserializedAttribute` custom attribute, the `System.Runtime.Serialization` namespace also defines `OnSerializingAttribute`, `OnSerializedAttribute`, and `OnDeserializingAttribute` custom attributes, which you can apply to your type's methods to have even more control over serialization and deserialization. Here is a sample class that applies each of these attributes to a method.

```
[Serializable]
public class MyType {
   Int32 x, y; [NonSerialized] Int32 sum;

   public MyType(Int32 x, Int32 y) {
      this.x = x; this.y = y; sum = x + y;
   }

   [OnDeserializing]
   private void OnDeserializing(StreamingContext context) {
      // Example: Set default values for fields in a new version of this type
   }

   [OnDeserialized]
   private void OnDeserialized(StreamingContext context) {
      // Example: Initialize transient state from fields
      sum = x + y;
   }

   [OnSerializing]
   private void OnSerializing(StreamingContext context) {
```

[2] Use of the `System.Runtime.Serialization.OnDeserialized` custom attribute is the preferred way of invoking a method when an object is deserialized, as opposed to having a type implement the `System.Runtime.Serialization.IDeserializationCallback` interface's `OnDeserialization` method.

```
      // Example: Modify any state before serializing
   }

   [OnSerialized]
   private void OnSerialized(StreamingContext context) {
      // Example: Restore any state after serializing
   }
}
```

Whenever you use any of these four attributes, the method you define must take a single `StreamingContext` parameter (discussed in the "Streaming Contexts" section later in this chapter) and return `void`. The name of the method can be anything you want it to be. Also, you should declare the method as `private` to prevent it from being called by normal code; the formatters run with enough security that they can call private methods.

Note When you are serializing a set of objects, the formatter first calls all of the objects' methods that are marked with the `OnSerializing` attribute. Next, it serializes all of the objects' fields, and finally it calls all of the objects' methods marked with the `OnSerialized` attribute. Similarly, when you deserialize a set of objects, the formatter calls all of the objects' methods that are marked with the `OnDeserializing` attribute, then it deserializes all of the object's fields, and then it calls all of the objects' methods marked with the `OnDeserialized` attribute.

Note also that during deserialization, when a formatter sees a type offering a method marked with the `OnDeserialized` attribute, the formatter adds this object's reference to an internal list. After all the objects have been deserialized, the formatter traverses this list in reverse order and calls each object's `OnDeserialized` method. When this method is called, all the serializable fields will be set correctly, and they may be accessed to perform any additional work that would be necessary to fully deserialize the object. Invoking these methods in reverse order is important because it allows inner objects to finish their deserialization before the outer objects that contain them finish their deserialization.

For example, imagine a collection object (like `Hashtable` or `Dictionary`) that internally uses a hash table to maintain its sets of items. The collection object type would implement a method marked with the `OnDeserialized` attribute. Even though the collection object would start being deserialized first (before its items), its `OnDeserialized` method would be called last (after any of its items' `OnDeserialized` methods). This allows the items to complete deserialization so that all their fields are initialized properly, allowing a good hash code value to be calculated. Then, the collection object creates its internal buckets and uses the items' hash codes to place the items into the buckets. I show an example of how the `Dictionary` class uses this in the upcoming "Controlling the Serialized/Deserialized Data" section of this chapter.

If you serialize an instance of a type, add a new field to the type, and then try to deserialize the object that did not contain the new field, the formatter throws a `SerializationException` with a message indicating that the data in the stream being deserialized has the wrong number of members. This is very problematic in versioning scenarios where it is common to add new fields to a type in a newer version. Fortunately, you can use the `System.Runtime.Serialization.OptionalFieldAttribute` attribute to help you.

You apply the `OptionalFieldAttribute` attribute to each new field you add to a type. Now, when the formatters see this attribute applied to a field, the formatters will not throw the `SerializationException` exception if the data in the stream does not contain the field.

How Formatters Serialize Type Instances

In this section, I give a bit more insight into how a formatter serializes an object's fields. This knowledge can help you understand the more advanced serialization and deserialization techniques explained in the remainder of this chapter.

To make things easier for a formatter, the FCL offers a `FormatterServices` type in the `System.Runtime.Serialization` namespace. This type has only static methods in it, and no instances of the type may be instantiated. The following steps describe how a formatter automatically serializes an object whose type has the `SerializableAttribute` attribute applied to it.

1. The formatter calls `FormatterServices`'s `GetSerializableMembers` method.

   ```
   public static MemberInfo[] GetSerializableMembers(Type type, StreamingContext context);
   ```

   This method uses reflection to get the type's public and private instance fields (excluding any fields marked with the `NonSerializedAttribute` attribute). The method returns an array of `MemberInfo` objects, one for each serializable instance field.

2. The object being serialized and the array of `System.Reflection.MemberInfo` objects are then passed to `FormatterServices`' static `GetObjectData` method.

   ```
   public static Object[] GetObjectData(Object obj, MemberInfo[] members);
   ```

   This method returns an array of `Objects` where each element identifies the value of a field in the object being serialized. This `Object` array and the `MemberInfo` array are parallel. That is, element 0 in the `Object` array is the value of the member identified by element 0 in the `MemberInfo` array.

3. The formatter writes the assembly's identity and the type's full name to the stream.

4. The formatter then enumerates over the elements in the two arrays, writing each member's name and value to the stream.

The following steps describe how a formatter automatically deserializes an object whose type has the `SerializableAttribute` attribute applied to it:

1. The formatter reads the assembly's identity and full type name from the stream. If the assembly is not currently loaded into the AppDomain, it is loaded (as described earlier). If the assembly can't be loaded, a `SerializationException` exception is thrown and the object cannot be deserialized. If the assembly is loaded, the formatter passes the assembly identity information and the type's full name to `FormatterServices`' static `GetTypeFromAssembly` method.

   ```
   public static Type GetTypeFromAssembly(Assembly assem, String name);
   ```

   This method returns a `System.Type` object indicating the type of object that is being deserialized.

2. The formatter calls `FormatterServices`'s static `GetUninitializedObject` method.

   ```
   public static Object GetUninitializedObject(Type type);
   ```

   This method allocates memory for a new object but does not call a constructor for the object. However, all the object's bytes are initialized to `null` or 0.

3. The formatter now constructs and initializes a `MemberInfo` array as it did before by calling the `FormatterServices`'s `GetSerializableMembers` method. This method returns the set of fields that were serialized and that need to be deserialized.

4. The formatter creates and initializes an `Object` array from the data contained in the stream.

5. The reference to the newly allocated object, the `MemberInfo` array, and the parallel `Object` array of field values is passed to `FormatterServices`' static `PopulateObjectMembers` method.

   ```
   public static Object PopulateObjectMembers(
      Object obj, MemberInfo[] members, Object[] data);
   ```

   This method enumerates over the arrays, initializing each field to its corresponding value. At this point, the object has been completely deserialized.

Controlling the Serialized/Deserialized Data

As discussed earlier in this chapter, the best way to get control over the serialization and deserialization process is to use the `OnSerializing`, `OnSerialized`, `OnDeserializing`, `OnDeserialized`, `NonSerialized`, and `OptionalField` attributes. However, there are some very rare scenarios where these attributes do not give you all the control you need. In addition, the formatters use reflection internally and reflection is slow, which increases the time it takes to serialize and deserialize objects. To get complete control over what data is serialized/deserialized or to eliminate the use of reflection,

your type can implement the System.Runtime.Serialization.ISerializable interface, which is defined as follows.

```
public interface ISerializable {
   void GetObjectData(SerializationInfo info, StreamingContext context);
}
```

This interface has just one method in it, GetObjectData. But most types that implement this interface will also implement a special constructor that I'll describe shortly.

Important The big problem with the ISerializable interface is that after a type implements it, all derived types must implement it too, and the derived types must make sure that they invoke the base class's GetObjectData method and the special constructor. In addition, after a type implements this interface, it can never remove it because it will lose compatibility with the derived types. It is always OK for sealed types to implement the ISerializable interface. Using the custom attributes described earlier in this chapter avoids all of the potential problems associated with the ISerializable interface.

Important The ISerializable interface and the special constructor are intended to be used by the formatters. However, other code could call GetObjectData, which might then return potentially sensitive information, or other code could construct an object that passes in corrupt data. For this reason, it is recommended that you apply the following attribute to the GetObjectData method and the special constructor.

```
[SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)]
```

When a formatter serializes an object graph, it looks at each object. If its type implements the ISerializable interface, then the formatter ignores all custom attributes and instead constructs a new System.Runtime.Serialization.SerializationInfo object. This object contains the actual set of values that should be serialized for the object.

When constructing a SerializationInfo, the formatter passes two parameters: Type and System.Runtime.Serialization.IFormatterConverter. The Type parameter identifies the object that is being serialized. Two pieces of information are required to uniquely identify a type: the string name of the type and its assembly's identity (which includes the assembly name, version, culture, and public key). When a SerializationInfo object is constructed, it obtains the type's full name (by internally querying Type's FullName property) and stores this string in a private field. You can obtain the type's full name by querying SerializationInfo's FullTypeName property. Likewise, the constructor obtains the type's defining assembly (by internally querying Type's Module property followed by querying Module's Assembly property followed by querying Assembly's FullName property) and stores this string in a private field. You can obtain the assembly's identity by querying SerializationInfo's AssemblyName property.

Note Although you can set a `SerializationInfo`'s `FullTypeName` and `AssemblyName` properties, this is discouraged. If you want to change the type that is being serialized, it is recommended that you call `SerializationInfo`'s `SetType` method, passing a reference to the desired `Type` object. Calling `SetType` ensures that the type's full name and defining assembly are set correctly. An example of calling `SetType` is shown in the "Serializing a Type As a Different Type and Deserializing an Object As a Different Object" section later in this chapter.

After the `SerializationInfo` object is constructed and initialized, the formatter calls the type's `GetObjectData` method, passing it the reference to the `SerializationInfo` object. The `GetObjectData` method is responsible for determining what information is necessary to serialize the object and adding this information to the `SerializationInfo` object. `GetObjectData` indicates what information to serialize by calling one of the many overloaded `AddValue` methods provided by the `SerializationInfo` type. `AddValue` is called once for each piece of data that you want to add.

The following code shows an approximation of how the `Dictionary<TKey, TValue>` type implements the `ISerializable` and `IDeserializationCallback` interfaces to take control over the serialization and deserialization of its objects.

```
[Serializable]
public class Dictionary<TKey, TValue>: ISerializable, IDeserializationCallback {
   // Private fields go here (not shown)

   private SerializationInfo m_siInfo;  // Only used for deserialization

   // Special constructor (required by ISerializable) to control deserialization
   [SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)]
   protected Dictionary(SerializationInfo info, StreamingContext context) {
      // During deserialization, save the SerializationInfo for OnDeserialization
      m_siInfo = info;
   }

   // Method to control serialization
   [SecurityCritical]
   public virtual void GetObjectData(SerializationInfo info, StreamingContext context) {

      info.AddValue("Version", m_version);
      info.AddValue("Comparer", m_comparer, typeof(IEqualityComparer<TKey>));
      info.AddValue("HashSize", (m_ buckets == null) ? 0 : m_buckets.Length);
      if (m_buckets != null) {
         KeyValuePair<TKey, TValue>[] array = new KeyValuePair<TKey, TValue>[Count];
         CopyTo(array, 0);
         info.AddValue("KeyValuePairs", array, typeof(KeyValuePair<TKey, TValue>[]));
      }
   }
```

```
   // Method called after all key/value objects have been deserialized
   public virtual void IDeserializationCallback.OnDeserialization(Object sender) {
      if (m_siInfo == null) return; // Never set, return

      Int32 num = m_siInfo.GetInt32("Version");
      Int32 num2 = m_siInfo.GetInt32("HashSize");
      m_comparer = (IEqualityComparer<TKey>)
         m_siInfo.GetValue("Comparer", typeof(IEqualityComparer<TKey>));
      if (num2 != 0) {
         m_buckets = new Int32[num2];
         for (Int32 i = 0; i < m_buckets.Length; i++) m_buckets[i] = -1;
         m_entries = new Entry<TKey, TValue>[num2];
         m_freeList = -1;
         KeyValuePair<TKey, TValue>[] pairArray = (KeyValuePair<TKey, TValue>[])
            m_siInfo.GetValue("KeyValuePairs", typeof(KeyValuePair<TKey, TValue>[]));
         if (pairArray == null)
            ThrowHelper.ThrowSerializationException(
               ExceptionResource.Serialization_MissingKeys);

         for (Int32 j = 0; j < pairArray.Length; j++) {
            if (pairArray[j].Key == null)
               ThrowHelper.ThrowSerializationException(
                  ExceptionResource.Serialization_NullKey);

            Insert(pairArray[j].Key, pairArray[j].Value, true);
         }
      } else { m_buckets = null; }
      m_version = num;
      m_siInfo = null;
}
```

Each `AddValue` method takes a `String` name and some data. Usually, the data is of a simple value type like `Boolean`, `Char`, `Byte`, `SByte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Single`, `Double`, `Decimal`, or `DateTime`. However, you can also call `AddValue`, passing it a reference to an `Object` such as a `String`. After `GetObjectData` has added all of the necessary serialization information, it returns to the formatter.

Note You should *always* call one of the overloaded `AddValue` methods to add serialization information for your type. If a field's type implements the `ISerializable` interface, don't call the `GetObjectData` on the field. Instead, call `AddValue` to add the field; the formatter will see that the field's type implements `ISerializable` and the formatter will call `GetObjectData` for you. If you were to call `GetObjectData` on the field object, the formatter wouldn't know to create a new object when deserializing the stream.

The formatter now takes all of the values added to the SerializationInfo object and serializes each of them out to the stream. You'll notice that the GetObjectData method is passed another parameter: a reference to a System.Runtime.Serialization.StreamingContext object. Most types' GetObjectData methods will completely ignore this parameter, so I will not discuss it now. Instead, I'll discuss it in the "Streaming Contexts" section later in this chapter.

So now you know how to set all of the information used for serialization. At this point, let's turn our attention to deserialization. As the formatter extracts an object from the stream, it allocates memory for the new object (by calling the System.Runtime.Serialization.FormatterServices type's static GetUninitializedObject method). Initially, all of this object's fields are set to 0 or null. Then, the formatter checks if the type implements the ISerializable interface. If this interface exists, the formatter attempts to call a special constructor whose parameters are identical to that of the GetObjectData method.

If your class is sealed, then it is highly recommended that you declare this special constructor to be private. This will prevent any code from accidentally calling increasing security. If not, then you should declare this special constructor as protected so that only derived classes can call it. Note that the formatters are able to call this special constructor no matter how it is declared.

This constructor receives a reference to a SerializationInfo object containing all of the values added to it when the object was serialized. The special constructor can call any of the GetBoolean, GetChar, GetByte, GetSByte, GetInt16, GetUInt16, GetInt32, GetUInt32, GetInt64, GetUInt64, GetSingle, GetDouble, GetDecimal, GetDateTime, GetString, and GetValue methods, passing in a string corresponding to the name used to serialize a value. The value returned from each of these methods is then used to initialize the fields of the new object.

When deserializing an object's fields, you should call the Get method that matches the type of value that was passed to the AddValue method when the object was serialized. In other words, if the GetObjectData method called AddValue, passing it an Int32 value, then the GetInt32 method should be called for the same value when deserializing the object. If the value's type in the stream doesn't match the type you're trying to get, then the formatter will attempt to use an IFormatterConvert object to "cast" the stream's value to the desired type.

As I mentioned earlier, when a SerializationInfo object is constructed, it is passed an object whose type implements the IFormatterConverter interface. Because the formatter is responsible for constructing the SerializationInfo object, it chooses whatever IFormatterConverter type it wants. Microsoft's BinaryFormatter and SoapFormatter types always construct an instance of the System.Runtime.Serialization.FormatterConverter type. Microsoft's formatters don't offer any way for you to select a different IFormatterConverter type.

The FormatterConverter type calls the System.Convert class's static methods to convert values between the core types, such as converting an Int32 to an Int64. However, to convert a value between other arbitrary types, the FormatterConverter calls Convert's ChangeType method to

cast the serialized (or original) type to an `IConvertible` interface and then calls the appropriate interface method. Therefore, to allow objects of a serializable type to be deserialized as a different type, you may want to consider having your type implement the `IConvertible` interface. Note that the `FormatterConverter` object is used only when deserializing objects and when you're calling a `Get` method whose type doesn't match the type of the value in the stream.

Instead of calling the various `Get` methods previously listed, the special constructor could instead call `GetEnumerator`, which returns a `System.Runtime.Serialization.SerializationInfoEnumerator` object that can be used to iterate through all the values contained within the `SerializationInfo` object. Each value enumerated is a `System.Runtime.Serialization.SerializationEntry` object.

Of course, you are welcome to define a type of your own that derives from a type that implements `ISerializable`'s `GetObjectData` and special constructor. If your type also implements `ISerializable`, then your implementation of `GetObjectData` and your implementation of the special constructor must call the same functions in the base class in order for the object to be serialized and deserialized properly. Do not forget to do this or the objects will not serialize or deserialize correctly. The next section explains how to properly define an `ISerializable` type whose base type doesn't implement this interface.

If your derived type doesn't have any additional fields in it and therefore has no special serialization/deserialization needs, then you do not have to implement `ISerializable` at all. Like all interface members, `GetObjectData` is virtual and will be called to properly serialize the object. In addition, the formatter treats the special constructor as "virtualized." That is, during deserialization, the formatter will check the type that it is trying to instantiate. If that type doesn't offer the special constructor, then the formatter will scan base classes until it finds one that implements the special constructor.

Important The code in the special constructor typically extracts its fields from the `SerializationInfo` object that is passed to it. As the fields are extracted, you are not guaranteed that the objects are fully deserialized, so the code in the special constructor should not attempt to manipulate the objects that it extracts.

If your type must access members (such as call methods) on an extracted object, then it is recommended that your type also provide a method that has the `OnDeserialized` attribute applied to it or have your type implement the `IDeserializationCallback` interface's `OnDeserialization` method (as shown in the `Dictionary` example). When this method is called, all objects have had their fields set. However, there is no guarantee to the order in which multiple objects have their `OnDeserialized` or `OnDeserialization` method called. So, although the fields may be initialized, you still don't know if a referenced object is completely deserialized if that referenced object also provides an `OnDeserialized` method or implements the `IDeserializationCallback` interface.

How to Define a Type That Implements ISerializable When the Base Type Doesn't Implement This Interface

As mentioned earlier, the ISerializable interface is extremely powerful, because it allows a type to take complete control over how instances of the type get serialized and deserialized. However, this power comes at a cost: The type is now responsible for serializing all of its base type's fields as well. Serializing the base type's fields is easy if the base type also implements the ISerializable interface; you just call the base type's GetObjectData method.

However, someday, you may find yourself defining a type that needs to take control of its serialization, but whose base type does not implement the ISerializable interface. In this case, your derived class must manually serialize the base type's fields by grabbing their values and adding them to the SerializationInfo collection. Then, in your special constructor, you will also have to get the values out of the collection and somehow set the base class's fields. Doing all of this is easy (albeit tedious) if the base class's fields are public or protected, but it can be very difficult or impossible to do if the base class's fields are private.

This following code shows how to properly implement ISerializable's GetObjectData method and its implied constructor so that the base type's fields are serialized.

```
[Serializable]
internal class Base {
   protected String m_name = "Jeff";
   public Base() { /* Make the type instantiable */ }
}

[Serializable]
internal sealed class Derived : Base, ISerializable {
   private DateTime m_date = DateTime.Now;
   public Derived() { /* Make the type instantiable*/ }

   // If this constructor didn't exist, we'd get a SerializationException
   // This constructor should be protected if this class were not sealed
   [SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)]
   private Derived(SerializationInfo info, StreamingContext context) {
      // Get the set of serializable members for our class and base classes
      Type baseType = this.GetType().BaseType;
      MemberInfo[] mi = FormatterServices.GetSerializableMembers(baseType, context);

      // Deserialize the base class's fields from the info object
      for (Int32 i = 0; i < mi.Length; i++) {
         // Get the field and set it to the deserialized value
         FieldInfo fi = (FieldInfo)mi[i];
         fi.SetValue(this, info.GetValue(baseType.FullName + "+" + fi.Name, fi.FieldType));
      }

      // Deserialize the values that were serialized for this class
      m_date = info.GetDateTime("Date");
   }
```

```
   [SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)]
   public virtual void GetObjectData(SerializationInfo info, StreamingContext context) {
      // Serialize the desired values for this class
      info.AddValue("Date", m_date);

      // Get the set of serializable members for our class and base classes
      Type baseType = this.GetType().BaseType;
      MemberInfo[] mi = FormatterServices.GetSerializableMembers(baseType, context);

      // Serialize the base class's fields to the info object
      for (Int32 i = 0; i < mi.Length; i++) {
         // Prefix the field name with the fullname of the base type
         info.AddValue(baseType.FullName + "+" + mi[i].Name,
            ((FieldInfo)mi[i]).GetValue(this));
      }
   }
   public override String ToString() {
      return String.Format("Name={0}, Date={1}", m_name, m_date);
   }
}
```

In this code, there is a base class, `Base`, which is marked only with the `SerializableAttribute` custom attribute. Derived from `Base` is `Derived`, which also is marked with the `SerializableAttribute` attribute and also implements the `ISerializable` interface. To make the situation more interesting, you'll notice that both classes define a `String` field called `m_name`. When calling `SerializationInfo`'s `AddValue` method, you can't add multiple values with the same name. The preceding code handles this situation by identifying each field by its class name prepended to the field's name. For example, when the `GetObjectData` method calls `AddValue` to serialize `Base`'s `m_name` field, the name of the value is written as "Base+m_name."

Streaming Contexts

As mentioned earlier, there are many destinations for a serialized set of objects: same process, different process on the same machine, different process on a different machine, and so on. In some rare situations, an object might want to know where it is going to be deserialized so that it can emit its state differently. For example, an object that wraps a Windows semaphore object might decide to serialize its kernel handle if the object knows that it will be deserialized into the same process, because kernel handles are valid within a process. However, the object might decide to serialize the semaphore's string name if it knows that the object will be deserialized on the same machine but into a different process. Finally, the object might decide to throw an exception if it knows that it will be deserialized in a process running on a different machine because a semaphore is valid only within a single machine.

A number of the methods mentioned earlier in this chapter accept a `StreamingContext`. A `StreamingContext` structure is a very simple value type offering just two public read-only properties, as shown in Table 24-1.

TABLE 24-1 StreamingContext's Public Read-Only Properties

Member Name	Member Type	Description
State	StreamingContextStates	A set of bit flags indicating the source or destination of the objects being serialized/deserialized
Context	Object	A reference to an object that contains any user-desired context information

A method that receives a StreamingContext structure can examine the State property's bit flags to determine the source or destination of the objects being serialized/deserialized. Table 24-2 shows the possible bit flag values.

TABLE 24-2 StreamingContextStates's Flags

Flag Name	Flag Value	Description
CrossProcess	0x0001	The source or destination is a different process on the same machine.
CrossMachines	0x0002	The source or destination is on a different machine.
File	0x0004	The source or destination is a file. Don't assume that the same process will deserialize the data.
Persistence	0x0008	The source or destination is a store such as a database or a file. Don't assume that the same process will deserialize the data.
Remoting	0x0010	The source or destination is remoting to an unknown location. The location may be on the same machine but may also be on another machine.
Other	0x0020	The source or destination is unknown.
Clone	0x0040	The object graph is being cloned. The serialization code may assume that the same process will deserialize the data, and it is therefore safe to access handles or other unmanaged resources.
CrossAppDomain	0x0080	The source or destination is a different AppDomain.
All	0x00FF	The source or destination may be any of the above contexts. This is the default context.

Now that you know how to get this information, let's discuss how you would set this information. The IFormatter interface (which is implemented by both the BinaryFormatter and the SoapFormatter types) defines a read/write StreamingContext property called Context. When you construct a formatter, the formatter initializes its Context property so that StreamingContextStates is set to All and the reference to the additional state object is set to null.

After the formatter is constructed, you can construct a StreamingContext structure using any of the StreamingContextStates bit flags, and you can optionally pass a reference to an object containing any additional context information you need. Now, all you need to do is set the formatter's Context property with this new StreamingContext object before calling the formatter's Serialize or Deserialize methods. Code demonstrating how to tell a formatter that you are serializing/deserialzing an object graph for the sole purpose of cloning all the objects in the graph is shown in the DeepClone method presented earlier in this chapter.

Serializing a Type As a Different Type and Deserializing an Object As a Different Object

The .NET Framework's serialization infrastructure is quite rich, and in this section, we discuss how a developer can design a type that can serialize or deserialize itself into a different type or object. Below are some examples where this is interesting:

- Some types (such as `System.DBNull` and `System.Reflection.Missing`) are designed to have only one instance per AppDomain. These types are frequently called *singletons*. If you have a reference to a `DBNull` object, serializing and deserializing it should not cause a new `DBNull` object to be created in the AppDomain. After deserializing, the returned reference should refer to the AppDomain's already-existing `DBNull` object.

- Some types (such as `System.Type`, `System.Reflection.Assembly`, and other reflection types like `MemberInfo`) have one instance per type, assembly, member, and so on. Imagine you have an array where each element references a `MemberInfo` object. It's possible that five array elements reference a single `MemberInfo` object. After serializing and deserializing this array, the five elements that referred to a single `MemberInfo` object should all refer to a single `MemberInfo` object. What's more, these elements should refer to the one `MemberInfo` object that exists for the specific member in the AppDomain. You could also imagine how this could be useful for polling database connection objects or any other type of object.

- For remotely controlled objects, the CLR serializes information about the server object that, when deserialized on the client, causes the CLR to create a proxy object. This type of the proxy object is a different type than the server object, but this is transparent to the client code. When the client calls instance methods on the proxy object, the proxy code internally remotes the call to the server that actually performs the request.

Let's look at some code that shows how to properly serialize and deserialize a singleton type.

```
// There should be only one instance of this type per AppDomain
[Serializable]
public sealed class Singleton : ISerializable {
   // This is the one instance of this type
   private static readonly Singleton s_theOneObject = new Singleton();

   // Here are the instance fields
   public String Name = "Jeff";
   public DateTime Date = DateTime.Now;

   // Private constructor allowing this type to construct the singleton
   private Singleton() { }

   // Method returning a reference to the singleton
   public static Singleton GetSingleton() { return s_theOneObject; }
```

```
   // Method called when serializing a Singleton
   // I recommend using an Explicit Interface Method Impl. Here
   [SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)]
   void ISerializable.GetObjectData(SerializationInfo info, StreamingContext context) {
      info.SetType(typeof(SingletonSerializationHelper));
      // No other values need to be added
   }

   [Serializable]
   private sealed class SingletonSerializationHelper : IObjectReference {
      // Method called after this object (which has no fields) is deserialized
      public Object GetRealObject(StreamingContext context) {
         return Singleton.GetSingleton();
      }
   }

   // NOTE: The special constructor is NOT necessary because it's never called
}
```

The `Singleton` class represents a type that allows only one instance of itself to exist per AppDomain. The following code tests the `Singleton`'s serialization and deserialization code to ensure that only one instance of the `Singleton` type ever exists in the AppDomain.

```
private static void SingletonSerializationTest() {
   // Create an array with multiple elements referring to the one Singleton object
   Singleton[] a1 = { Singleton.GetSingleton(), Singleton.GetSingleton() };
   Console.WriteLine("Do both elements refer to the same object? "
      + (a1[0] == a1[1])); // "True"

   using (var stream = new MemoryStream()) {
      BinaryFormatter formatter = new BinaryFormatter();

      // Serialize and then deserialize the array elements
      formatter.Serialize(stream, a1);
      stream.Position = 0;
      Singleton[] a2 = (Singleton[])formatter.Deserialize(stream);

      // Prove that it worked as expected:
      Console.WriteLine("Do both elements refer to the same object? "
         + (a2[0] == a2[1])); // "True"
      Console.WriteLine("Do all  elements refer to the same object? "
         + (a1[0] == a2[0])); // "True"
   }
}
```

Now, let's walk through the code to understand what's happening. When the `Singleton` type is loaded into the AppDomain, the CLR calls its static constructor, which constructs a `Singleton` object and saves a reference to it in a static field, `s_theOneObject`. The `Singleton` class doesn't offer any public constructors, which prevents any other code from constructing any other instances of this class.

In SingletonSerializationTest, an array is created consisting of two elements; each element references the Singleton object. The two elements are initialized by calling Singleton's static GetSingleton method. This method returns a reference to the one Singleton object. The first call to Console's WriteLine method displays "True," verifying that both array elements refer to the same exact object.

Now, SingletonSerializationTest calls the formatter's Serialize method to serialize the array and its elements. When serializing the first Singleton, the formatter detects that the Singleton type implements the ISerializable interface and calls the GetObjectData method. This method calls SetType, passing in the SingletonSerializationHelper type, which tells the formatter to serialize the Singleton object as a SingletonSerializationHelper object instead. Because AddValue is not called, no additional field information is written to the stream. Because the formatter automatically detected that both array elements refer to a single object, the formatter serializes only one object.

After serializing the array, SingletonSerializationTest calls the formatter's Deserialize method. When deserializing the stream, the formatter tries to deserialize a SingletonSerializationHelper object because this is what the formatter was "tricked" into serializing. (In fact, this is why the Singleton class doesn't provide the special constructor that is usually required when implementing the ISerializable interface.) After constructing the SingletonSerializationHelper object, the formatter sees that this type implements the System.Runtime.Serialization.IObjectReference interface. This interface is defined in the FCL as follows.

```
public interface IObjectReference {
   Object GetRealObject(StreamingContext context);
}
```

When a type implements this interface, the formatter calls the GetRealObject method. This method returns a reference to the object that you really want a reference to now that deserialization of the object has completed. In my example, the SingletonSerializationHelper type has GetRealObject return a reference to the Singleton object that already exists in the AppDomain. So, when the formatter's Deserialize method returns, the a2 array contains two elements, both of which refer to the AppDomain's Singleton object. The SingletonSerializationHelper object used to help with the deserialization is immediately unreachable and will be garbage collected in the future.

The second call to WriteLine displays "True," verifying that both of a2's array elements refer to the exact same object. The third and last call to WriteLine also displays "True," proving that the elements in both arrays all refer to the exact same object.

Serialization Surrogates

Up to now, I've been discussing how to modify a type's implementation to control how a type serializes and deserializes instances of itself. However, the formatters also allow code that is not part of the type's implementation to override how a type serializes and deserializes its objects. There are two main reasons why application code might want to override a type's behavior:

- It allows a developer the ability to serialize a type that was not originally designed to be serialized.
- It allows a developer to provide a way to map one version of a type to a different version of a type.

Basically, to make this mechanism work, you first define a "surrogate type" that takes over the actions required to serialize and deserialize an existing type. Then, you register an instance of your surrogate type with the formatter telling the formatter which existing type your surrogate type is responsible for acting on. When the formatter detects that it is trying to serialize or deserialize an instance of the existing type, it will call methods defined by your surrogate object. Let's build a sample that demonstrates how all this works.

A serialization surrogate type must implement the `System.Runtime.Serialization.ISerializationSurrogate` interface, which is defined in the FCL as follows.

```
public interface ISerializationSurrogate {
   void GetObjectData(Object obj, SerializationInfo info, StreamingContext context);

   Object SetObjectData(Object obj, SerializationInfo info, StreamingContext context,
      ISurrogateSelector selector);
}
```

Now, let's walk through an example that uses this interface. Let's say your program contains some `DateTime` objects that contain values that are local to the user's computer. What if you want to serialize the `DateTime` objects to a stream but you want the values to be serialized in universal time? This would allow you to send the data over a network stream to another machine in another part of the world and have the `DateTime` value be correct. Although you can't modify the `DateTime` type that ships with the FCL, you can define your own serialization surrogate class that can control how `DateTime` objects are serialized and deserialized. Here is how to define the surrogate class.

```
internal sealed class UniversalToLocalTimeSerializationSurrogate : ISerializationSurrogate {
   public void GetObjectData(Object obj, SerializationInfo info, StreamingContext context) {
      // Convert the DateTime from local to UTC
      info.AddValue("Date", ((DateTime)obj).ToUniversalTime().ToString("u"));
   }

   public Object SetObjectData(Object obj, SerializationInfo info, StreamingContext context,
      ISurrogateSelector selector) {
      // Convert the DateTime from UTC to local
      return DateTime.ParseExact(info.GetString("Date"), "u", null).ToLocalTime();
   }
}
```

The GetObjectData method here works just like the ISerializable interface's GetObjectData method. The only difference is that ISerializationSurrogate's GetObjectData method takes one additional parameter: a reference to the "real" object that is to be serialized. In the GetObjectData method above, this object is cast to DateTime, the value is converted from local time to universal time, and a string (formatted using universal full date/time pattern) is added to the SerializationInfo collection.

The SetObjectData method is called in order to deserialize a DateTime object. When this method is called, it is passed a reference to a SerializationInfo object. SetObjectData gets the string date out of this collection, parses it as a universal full date/time formatted string, and then converts the resulting DateTime object from universal time to the machine's local time.

The Object that is passed for SetObjectData's first parameter is a bit strange. Just before calling SetObjectData, the formatter allocates (via FormatterServices's static GetUninitializedObject method) an instance of the type that the surrogate is a surrogate for. The instance's fields are all 0/null and no constructor has been called on the object. The code inside SetObjectData can simply initialize the fields of this instance by using the values from the passed-in SerializationInfo object and then have SetObjectData return null. Alternatively, SetObjectData could create an entirely different object or even a different type of object and return a reference to this new object, in which case, the formatter will ignore any changes that may or may not have happened to the object it passed in to SetObjectData.

In my example, my UniversalToLocalTimeSerializationSurrogate class acts as a surrogate for the DateTime type, which is a value type. And so, the obj parameter refers to a boxed instance of a DateTime. There is no way to change the fields in most value types (because they are supposed to be immutable) and so, my SetObjectData method ignores the obj parameter and returns a new DateTime object with the desired value in it.

At this point, I'm sure you're all wondering how the formatter knows to use this ISerializationSurrogate type when it tries to serialize/deserialize a DateTime object. The following code demonstrates how to test the UniversalToLocalTimeSerializationSurrogate class.

```
private static void SerializationSurrogateDemo() {
   using (var stream = new MemoryStream()) {
      // 1. Construct the desired formatter
      IFormatter formatter = new SoapFormatter();

      // 2. Construct a SurrogateSelector object
      SurrogateSelector ss = new SurrogateSelector();

      // 3. Tell the surrogate selector to use our surrogate for DateTime objects
      ss.AddSurrogate(typeof(DateTime), formatter.Context,
         new UniversalToLocalTimeSerializationSurrogate());

      // NOTE: AddSurrogate can be called multiple times to register multiple surrogates

      // 4. Tell the formatter to use our surrogate selector
      formatter.SurrogateSelector = ss;
```

```
      // Create a DateTime that represents the local time on the machine & serialize it
      DateTime localTimeBeforeSerialize = DateTime.Now;
      formatter.Serialize(stream, localTimeBeforeSerialize);

      // The stream displays the Universal time as a string to prove it worked
      stream.Position = 0;
      Console.WriteLine(new StreamReader(stream).ReadToEnd());

      // Deserialize the Universal time string & convert it to a local DateTime
      stream.Position = 0;
      DateTime localTimeAfterDeserialize = (DateTime)formatter.Deserialize(stream);

      // Prove it worked correctly:
      Console.WriteLine("LocalTimeBeforeSerialize ={0}", localTimeBeforeSerialize);
      Console.WriteLine("LocalTimeAfterDeserialize={0}", localTimeAfterDeserialize);
   }
}
```

After steps 1 through 4 have executed, the formatter is ready to use the registered surrogate types. When the formatter's `Serialize` method is called, each object's type is looked up in the set maintained by the `SurrogateSelector`. If a match is found, then the `ISerializationSurrogate` object's `GetObjectData` method is called to get the information that should be written out to the stream.

When the formatter's `Deserialize` method is called, the type of the object about to be deserialized is looked up in the formatter's `SurrogateSelector` and if a match is found, then the `ISerializationSurrogate` object's `SetObjectData` method is called to set the fields within the object being deserialized.

Internally, a `SurrogateSelector` object maintains a private hash table. When `AddSurrogate` is called, the `Type` and `StreamingContext` make up the key and the `ISerializationSurrogate` object is the key's value. If a key with the same `Type`/`StreamingContext` already exists, then `AddSurrogate` throws an `ArgumentException`. By including a `StreamingContext` in the key, you can register one surrogate type object that knows how to serialize/deserialize a `DateTime` object to a file and register a different surrogate object that knows how to serialize/deserialize a `DateTime` object to a different process.

Note The `BinaryFormatter` class has a bug that prevents a surrogate from serializing objects with references to each other. To fix this problem, you need to pass a reference to your `ISerializationSurrogate` object to `FormatterServices`'s static `GetSurrogateForCyclicalReference` method. This method returns an `ISerializationSurrogate` object, which you can then pass to the `SurrogateSelector`'s `AddSurrogate` method. However, when you use the `GetSurrogateForCyclicalReference` method, your surrogate's `SetObjectData` method must modify the value inside the object referred to by `SetObjectData`'s `obj` parameter and ultimately return `null` or `obj` to the calling method. The downloadable code that accompanies this book shows how to modify the `UniversalToLocalTimeSerializationSurrogate` class and the `SerializationSurrogateDemo` method to support cyclical references.

Surrogate Selector Chains

Multiple `SurrogateSelector` objects can be chained together. For example, you could have a `SurrogateSelector` that maintains a set of serialization surrogates that are used for serializing types into proxies that get remoted across the wire or between AppDomains. You could also have a separate `SurrogateSelector` object that contains a set of serialization surrogates that are used to convert Version 1 types into Version 2 types.

If you have multiple `SurrogateSelector` objects that you'd like the formatter to use, you must chain them together into a linked list. The `SurrogateSelector` type implements the `ISurrogateSelector` interface, which defines three methods. All three of these methods are related to chaining. Here is how the `ISurrogateSelector` interface is defined.

```
public interface ISurrogateSelector {
   void ChainSelector(ISurrogateSelector selector);
   ISurrogateSelector GetNextSelector();
   ISerializationSurrogate GetSurrogate(Type type, StreamingContext context,
      out ISurrogateSelector selector);
}
```

The `ChainSelector` method inserts an `ISurrogateSelector` object immediately after the `ISurrogateSelector` object being operated on ('`this`' object). The `GetNextSelector` method returns a reference to the next `ISurrogateSelector` object in the chain or `null` if the object being operated on is the end of the chain.

The `GetSurrogate` method looks up a `Type`/`StreamingContext` pair in the `ISurrogateSelector` object identified by `this`. If the pair cannot be found, then the next `ISurrogateSelector` object in the chain is accessed, and so on. If a match is found, then `GetSurrogate` returns the `ISerializationSurrogate` object that handles the serialization/deserialization of the type looked up. In addition, `GetSurrogate` also returns the `ISurrogateSelector` object that contained the match; this is usually not needed and is ignored. If none of the `ISurrogateSelector` objects in the chain have a match for the `Type`/`StreamingContext` pair, `GetSurrogate` returns `null`.

Note The FCL defines an `ISurrogateSelector` interface and also defines a `SurrogateSelector` type that implements this interface. However, it is extremely rare that anyone will ever have to define their own type that implements the `ISurrogateSelector` interface. The only reason to define your own type that implements this interface is if you need to have more flexibility over mapping one type to another. For example, you might want to serialize all types that inherit from a specific base class in a special way. The `System.Runtime.Remoting.Messaging.RemotingSurrogateSelector` class is a perfect example. When serializing objects for remoting purposes, the CLR formats the objects using the `RemotingSurrogateSelector`. This surrogate selector serializes all objects that derive from `System.MarshalByRefObject` in a special way so that deserialization causes proxy objects to be created on the client side.

Overriding the Assembly and/or Type When Deserializing an Object

When serializing an object, formatters output the type's full name and the full name of the type's defining assembly. When deserializing an object, formatters use this information to know exactly what type of object to construct and initialize. The earlier discussion about the `ISerialization-Surrogate` interface showed a mechanism allowing you to take over the serialization and deserialization duties for a specific type. A type that implements the `ISerializationSurrogate` interface is tied to a specific type in a specific assembly.

However, there are times when the `ISerializationSurrogate` mechanism doesn't provide enough flexibility. Here are some scenarios when it might be useful to deserialize an object into a different type than it was serialized as:

- A developer might decide to move a type's implementation from one assembly to a different assembly. For example, the assembly's version number changes making the new assembly different from the original assembly.
- An object on a server that gets serialized into a stream that is sent to a client. When the client processes the stream, it could deserialize the object to a completely different type whose code knows how to remotely invoke method calls to the server's object.
- A developer makes a new version of a type. We want to deserialize any already-serialized objects into the new version of the type.

The `System.Runtime.Serialization.SerializationBinder` class makes deserializing an object to a different type very easy. To do this, you first define your own type that derives from the abstract `SerializationBinder` type. In the following code, assume that version 1.0.0.0 of your assembly defined a class called `Ver1` and assume that the new version of your assembly defines the `Ver1ToVer2SerializationBinder` class and also defines a class called `Ver2`.

```
internal sealed class Ver1ToVer2SerializationBinder : SerializationBinder {
   public override Type BindToType(String assemblyName, String typeName) {
      // Deserialize any Ver1 object from version 1.0.0.0 into a Ver2 object

      // Calculate the assembly name that defined the Ver1 type
      AssemblyName assemVer1 = Assembly.GetExecutingAssembly().GetName();
      assemVer1.Version = new Version(1, 0, 0, 0);

      // If deserializing the Ver1 object from v1.0.0.0, turn it into a Ver2 object
      if (assemblyName == assemVer1.ToString() && typeName == "Ver1")
         return typeof(Ver2);

      // Else, just return the same type being requested
      return Type.GetType(String.Format("{0}, {1}", typeName, assemblyName));
   }
}
```

Now, after you construct a formatter, construct an instance of `Ver1ToVer2SerializationBinder` and set the formatter's `Binder` read/write property to refer to the binder object. After setting the `Binder` property, you can now call the formatter's `Deserialize` method. During deserialization, the formatter sees that a binder has been set. As each object is about to be deserialized, the formatter calls the binder's `BindToType` method, passing it the assembly name and type that the formatter wants to deserialize. At this point, `BindToType` decides what type should actually be constructed and returns this type.

Note The `SerializationBinder` class also makes it possible to change the assembly/type information while serializing an object by overriding its `BindToName` method, which looks like this.

```
public virtual void BindToName(Type serializedType,
   out string assemblyName, out string typeName)
```

During serialization, the formatter calls this method, passing you the type it wants to serialize. You can then return (via the two out parameters) the assembly and type that you want to serialize instead. If you return `null` and `null` (which is what the default implementation does), then no change is performed.