Module 6: Reading and Writing Local Data

Contents:

Module overview

Reading and writing data are core requirements for many applications, such as a text file editor saving a file to the file system, or a Windows service writing error messages to a custom log file. The local file system provides the perfect environment for an application to read and write such data, because access is fast and the location is readily available. The Microsoft .NET Framework provides a variety of I/O classes that simplify the process of implementing I/O functionality in your applications.

In this module, you will learn how to read and write data by using transactional file system I/O operations, how to serialize and deserialize data to the file system, and how to read and write data to the file system by using streams.

Objectives

After completing this module, you will be able to:

- Read and write data to and from the file system by using file I/O.

- Convert data into a format that can be written to or read from a file or other data source.
- Use streams to send and receive data to or from a file or data source

Lesson 1: Reading and Writing Files

The .NET Framework provides the **System.IO** namespace, which contains a number of classes that help simplify applications that require I/O functionality.

In this lesson, you will learn how to use the classes in this namespace to read and write data to and from files, and to manipulate files and directories on the file system.

Lesson objectives

After completing this lesson, you will be able to:

- Read and write data by using the **File** class.
- Manipulate files by using the **FileInfo** and the **File** classes.
- Manipulate directories by using the **DirectoryInfo** and **Directory** classes.
- Manipulate file and directory paths by using the **Path** class.

Reading and Writing Data by Using the File Class

- The **System.IO namespace** contains classes for manipulating files and directories
- The **File** class contains atomic read methods, including:
  - **ReadAllText(...)**
  - **ReadAllLines(...)**
- The **File** class contains atomic write methods, including:
  - **WriteAllText(...)**
  - **AppendAllText(...)**

The **File** class in the **System.IO** namespace exposes several static methods that you can use to perform transactional operations for direct reading and writing of files. These methods are transactional because they wrap several underlying functions into a single method call. Typically, to read data from a file, you:

1. Acquire the file handle.
2. Open a stream to the file.
3. Buffer the data from the file into memory.
4. Release the file handle so that it can be reused.

The static methods that the **File** class exposes are convenient because they encapsulate intricate, low-level functions. However their convenience and the fact that they shield the developer from the underlying functionality means in some cases they don't offer the control or flexibility that applications require. For example, the **ReadAllText** method will read the entire contents of a file into memory. For small files this will be fine, but for large files it can present scalability issues, and may result in an unresponsive UI in your application.

Reading Data from Files

The **File** class provides several methods that you can use to read data from a file. The format of your data and how your application intends to process it will influence the method that you should use. The following list describes some of these methods:

- The **ReadAllText** method enables you to read the contents of a file into a single string variable. The following code example shows how to read the contents of the settings.txt file into a string named **settings**.

```
string filePath = @"C:\fourthCoffee\settings.txt";
string settings = File.ReadAllText(filePath);
```

- The **ReadAllLines** method enables you to read the contents of a file and store each line at a new index in a string array. The following code example shows how to read the contents of the settings.txt file and store each line in the string array named **settingsLineByLine**.

```
string filePath = @"C:\fourthCoffee\settings.txt";
string[] settingsLineByLine = File.ReadAllLines(filePath);
```

- The **ReadAllBytes** method enables you to read the contents of a file as binary data and store the data in a byte array. The following code example shows how to read the contents of the settings.txt file into a byte array named **rawSettings**.

```
string filePath = @"C:\fourthCoffee\settings.txt";
byte[] rawSettings = File.ReadAllBytes(filePath);
```

> **Note:** During these examples and for the rest of this module, you'll see the notation **@""**. Normally, when the character '\' appears in a string, it's treated as an escape character, transforming the next character to a special Unicode or ASCII character. For example, the string "\n" will be transformed to the new line character. To write '\', you need to add another to escape it, like this: "\\".

String starting with **@**, called verbatim strings do not treat '\' as an **escape character**, and anything written in it will be treated literally. It's especially useful to shorten paths, negating the need for multiple "\\".
You can learn more about strings here: string (C# Reference), **https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/string**.

Each of these methods enables you to read the contents of a file into memory. You could use the **ReadAllText** method if you wanted to cache the entire file in memory in a single operation. Alternatively, if you wanted to process a file line-by-line, you could use the **ReadAllLines** method to read each line into an array.

Writing Data to Files

The **File** class also provides methods that you can use to write different types of data to a file. For each of the different types of data you can write, the **File** class provides two methods:

- If the specified file does not exist, the **Write***xxx* methods create a new file with the new data. If the file does exist, the **Write***xxx* methods overwrite the existing file with the new data.
- If the specified file does not exist, the **Append***xxx* methods also create a new file with the new data. However, if the file does exist, the new data is written to the end of the existing file.

The following list describes some of these methods:

- The **WriteAllText** method enables you to write the contents of a string variable to a file. If the file exists, its contents will be overwritten. The following code example shows how to write the contents of a string named **settings** to a new file named settings.txt.

```
string filePath = @"C:\fourthCoffee\settings.txt";
string settings = "companyName=fourth coffee;";
File.WriteAllText(filePath, settings);
```

- The **WriteAllLines** method enables you to write the contents of a string array to a file. Each entry in the string array represents a new line in the file. The following code example shows how to write the contents of a string array named **hosts** to a new file named hosts.txt.

```
string filePath = @"C:\fourthCoffee\hosts.txt ";
string[] hosts = { "86.120.1.203", "113.45.80.31",
"168.195.23.29" };
File.WriteAllLines(filePath, hosts);
```

- The **WriteAllBytes** method enables you to write the contents of a byte array to a binary file. The following code example shows how to write the contents of a byte array named **rawSettings** to a new file named settings.txt.

```
string filePath = @"C:\fourthCoffee\setting.txt ";
byte[] rawSettings =
{99,111,109,112,97,110,121,78,97,109,101,61,102,111,
                    117,114,116,104,32,99,111,102,102,101,101};
File.WriteAllBytes(filePath, rawSettings);
```

- The **AppendAllText** method enables you to write the contents of a string variable to the end of an existing file. The following code example shows how to write the contents of a string variable named **settings** to the end of the existing settings.txt file.

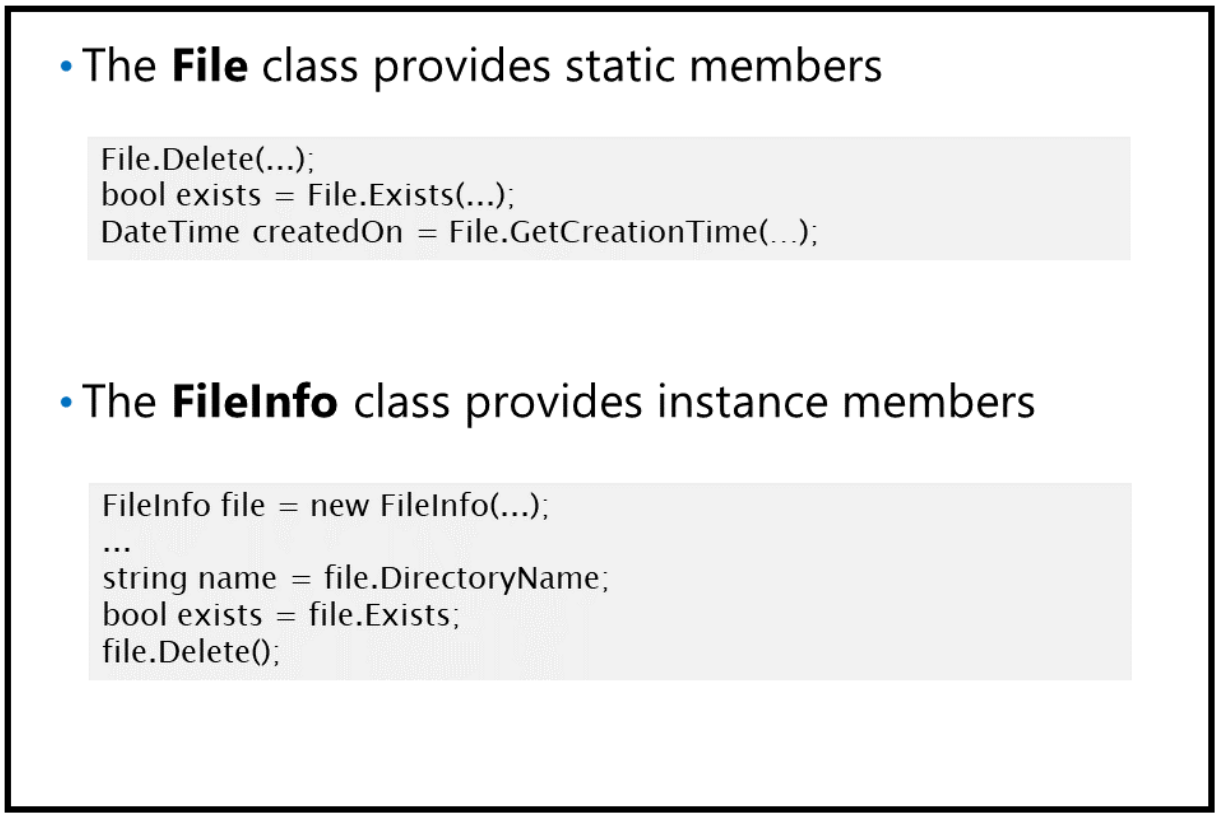
```
string filePath = @"C:\fourthCoffee\settings.txt";
string settings = "companyContact= Dean Halstead";
File.AppendAllText(filePath, settings);
```

- The **AppendAllLines** method enables you to write the contents of a string array to the end of an existing file. The following code example shows how to write the contents of a string array named **newHosts** to the existing hosts.txt file.

```
string filePath = @"C:\fourthCoffee\hosts.txt ";
string[] newHosts = { "97.11.1.195", "203.194.40.177" };
File.WriteAllLines(filePath, newHosts);
```

Each of these methods enables you to write data to a file. If you want to add data to an existing file that may already exist, then you should use an **Append***xxx* method. If you want to overwrite an existing file, then you should use a **Write***xxx* method. Then, depending on how you want the information is stored (whether as binary data, a textual blob in a string, or an array of strings representing each individual line) use the *xxx**AllBytes***, *xxx**AllText***, or *xxx**AllLines*** method.

Manipulating Files



As well as reading from and writing to files, applications typically require the ability to interact with files stored on the file system. For example, your application may need to copy a file from the system directory to a temporary location before performing some further processing, or your application may need to read some metadata associated with the file, such as the file creation time. You can implement this type of functionality by using the **File** and **FileInfo** classes.

File Manipulation by using the File Class

The **File** class provides static methods that you can use to perform basic file manipulation. The following list describes some of these methods:

- The **Copy** method enables you to copy an existing file to a different directory on the file system. The following code example shows how to copy the settings.txt file from the C:\fourthCoffee\ directory to the C:\temp\ directory.

```
string sourceSettingsPath = @"C:\fourthCoffee\settings.txt";
string destinationSettingsPath = @"C:\temp\settings.txt";
bool overwrite = true;
File.Copy(sourceSettingsPath, destinationSettingsPath,
overWrite);
```

> **Note:** The overwrite parameter passed to the **Copy** method call indicates that the copy process should overwrite an existing file if it exists at the destination path. If you pass **false** to the **Copy** method call, and the file already exists, the Common Language Runtime (CLR) will throw a **System.IO.IOException**.

- The **Delete** method enables you to delete an existing file from the file system. The following code example shows how to delete the existing settings.txt file.

```
string filePath = @"C:\fourthCoffee\settings.txt";
File.Delete(filePath);
```

- The **Exists** method enables you to check whether a file exists on the file system. The following code example shows how to check whether the settings.txt file exists.

```
string filePath = @"C:\fourthCoffee\settings.txt";
bool persistedSettingsExist = File.Exists(filePath);
```

- The **GetCreationTime** method enables you to read the date time stamp that

describes when a file was created, from the metadata associated with the file. The following code example shows how you can determine when the settings.txt file was created.

```
string filePath = @"C:\fourthCoffee\settings.txt";
DateTime settingsCreatedOn = File.GetCreationTime(filePath);
```

There are many other operations and metadata associated with files that you can utilize in your applications. The **FileInfo** class provides access to these through a number of instance members.

File Manipulation by using the FileInfo class

The **FileInfo** class provides instance members that you can use to manipulate an existing file. In contrast to the **File** class that provides static methods for direct manipulation, the **FileInfo** class behaves like an in-memory representation of the physical file, exposing metadata associated with the file through properties, and exposing operations through methods.

The following code example shows how to create an instance of the **FileInfo** class that represents the settings.txt file.

Instantiating the FileInfo Class

```
string filePath = @"C:\fourthCoffee\settings.txt";
FileInfo settings = new FileInfo(filePath);
```

After you have created an instance of the **FileInfo** class, you can use the properties and methods that it exposes to interact with the file. The following list describes some of these properties and methods:

- The **CopyTo** method enables you to copy an existing file to a different directory on the file system. The following code example shows how to copy the settings.txt file from the C:\fourthCoffee\ directory to the C:\temp\ directory.

```
string sourceSettingsPath = @"C:\fourthCoffee\settings.txt";
string destinationSettingsPath = @"C:\temp\settings.txt";
bool overwrite = true;
FileInfo settings = new FileInfo(sourceSettingsPath);
settings.CopyTo(destinationSettingsPath, overwrite);
```

> **Note:** The overwrite parameter passed to the **CopyTo** method call indicates that the copy process should overwrite an existing file if it exists at the destination path. If you pass **false** to the **CopyTo** method call, and the file already exists, the CLR will throw a **System.IO.IOException**.

- The **Delete** method enables you to delete a file. The following code example shows how to delete the settings.txt file.

```
string filePath = @"C:\fourthCoffee\settings.txt";
FileInfo settings = new FileInfo(filePath);
settings.Delete();
```

- The **DirectoryName** property enables you to get the directory path to the file. The following code example shows how to get the path to the settings.txt file.

```
string filePath = @"C:\fourthCoffee\settings.txt";
FileInfo settings = new FileInfo(filePath);
string directoryPath = settings.DirectoryName; // returns
C:\\fourthCoffee
```

- The **Exists** method enables you to determine if the file exists within the file system. The following code example shows how to check whether the settings.txt file exists.

```
string filePath = @"C:\fourthCoffee\settings.txt";
FileInfo settings = new FileInfo(filePath);
bool persistedSettingsExist = settings.Exists;
```

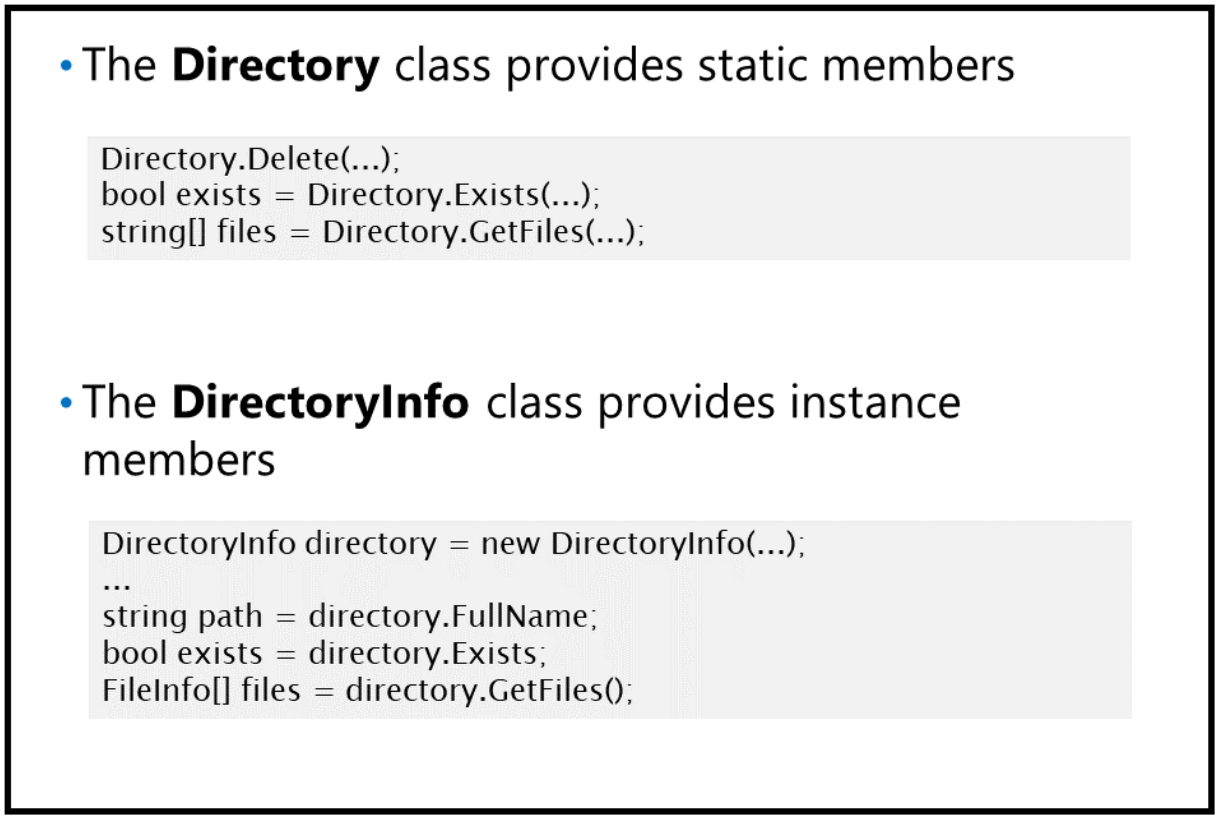
- The **Extension** property enables you to get the file extension of a file. The following code example shows how to get the extension of a path returned from a method call.

```
string filePath = FourthCoffeeDataService.GetDataPath();
FileInfo settings = new FileInfo(filePath);
string extension = settings.Extension;
```

- The **Length** property enables you to get the length of the file in bytes. The following code example shows how to get the length of the settings.txt file.

```
string filePath = @"C:\fourthCoffee\settings.txt";
FileInfo settings = new FileInfo(filePath);
long length = settings.Length;
```

Manipulating Directories



It is a common requirement for applications to interact and manipulate the file system directory structure, whether to check that a directory exists before writing a file or to remove directories when running a system cleanup process. The .NET Framework class library provides the **Directory** and **DirectoryInfo** classes for such operations.

Manipulating Directories by using the Directory Class

Similar to the **File** class, the **Directory** class provides static methods that enable you to interact with directories, without instantiating a directory-related object in your code. The following list describes some of these static methods:

- The **CreateDirectory** method enables you to create a new directory on the file system. The following example shows how to create the C:\fourthCoffee\tempData directory.

```
string directoryPath = @"C:\fourthCoffee\tempData";
Directory.CreateDirectory(directoryPath);
```

- The **Delete** method enables you to delete a directory at a specific path. The following code example shows how to delete the C:\fourthCoffee\tempData

directory, and all its contents.

```
string directoryPath = @"C:\fourthCoffee\tempData";
bool recursivelyDeleteSubContent = true;
Directory.Delete(directoryPath, recursivelyDeleteSubContent);
```

> **Note:** The **recursivelyDeleteSubContent** parameter passed into the **Delete** method call indicates whether the delete process should delete any content that may exist in the directory. If you pass **false** into the **Delete** method call, and the directory is not empty, the CLR will throw a **System.IO.IOException**.

- The **Exists** method enables you to determine if a directory exists on the file system. The following code example shows how to determine if the C:\fourthCoffee\tempData directory exists.

```
string directoryPath = @"C:\fourthCoffee\tempData";
bool tempDataDirectoryExists = Directory.Exists(directoryPath);
```

- The **GetDirectories** method enables you to get a list of all subdirectories within a specific directory on the file system. The following code example shows how to get a list of all the sub directories in the C:\fourthCoffee\tempData directory.

```
string directoryPath = @"C:\fourthCoffee\tempData";
string[] subDirectories =
Directory.GetDirectories(directoryPath);
```

- The **GetFiles** method enables you to get a list of all the files within a specific directory on the file system. The following example shows how to get a list of all the files in the C:\fourthCoffee\tempData directory.

```
string directoryPath = @"C:\fourthCoffee\tempData";
string[] files = Directory.GetFiles(directoryPath);
```

The **DirectoryInfo** class provides instance members that enable you to access directory metadata and manipulate the directory structure.

Manipulating Directories by using the DirectoryInfo Class

The **DirectoryInfo** class acts as an in-memory representation of a directory. Before you can access the properties and execute the methods that the **DirectoryInfo** class exposes, you must create an instance of the class.

The following code example shows how to create an instance of the **DirectoryInfo** class that represents the **C:\fourthCoffee\tempData** directory.

Instantiating the DirectoryInfo Class

```
string directoryPath = @"C:\fourthCoffee\tempData";
DirectoryInfo directory = new DirectoryInfo(directoryPath);
```

When you have created an instance of the **DirectoryInfo** class, you can then use its properties and methods to interact with the directory. The following list describes some of these properties and methods:

- The **Create** method enables you to create a new directory on the file system. The following example shows how to create the C:\fourthCoffee\tempData directory.

```
string directoryPath = @"C:\fourthCoffee\tempData";
DirectoryInfo directory = new DirectoryInfo(directoryPath);
directory.Create();
```

- The **Delete** method enables you to delete a directory at a specific path. The following code example shows how to delete the C:\fourthCoffee\tempData directory, and all its contents.

```
string directoryPath = @"C:\fourthCoffee\tempData";
bool recursivelyDeleteSubContent = true;
```

```
DirectoryInfo directory = new DirectoryInfo(directoryPath);
directory.Delete(recursivelyDeleteSubContent);
```

> **Note:** The **recursivelyDeleteSubContent** parameter passed to the **Delete** method call indicates whether the delete process should delete any content that may exist in the directory. If you pass **false** to the **Delete** method call, and the directory is not empty, the CLR will throw a **System.IO.IOException**.

- The **Exists** property enables you to determine if a directory exists on the file system. The following code example shows how to determine if the C:\fourthCoffee\tempData directory exists.

```
string directoryPath = @"C:\fourthCoffee\tempData";
DirectoryInfo directory = new DirectoryInfo(directoryPath);
bool tempDataDirectoryExists = directory.Exists;
```

- The **FullName** property enables you to get the full path to the directory. The following example shows how to get the full path to the tempData directory.

```
string directoryPath = @"C:\fourthCoffee\tempData";
DirectoryInfo directory = new DirectoryInfo(directoryPath);
string fullPath = directory.FullName;
```

- The **GetDirectories** method enables you to get a list of all subdirectories within a specific directory on the file system. In contrast to the static **File.GetDirectories** method, this instance method returns an array of type **DirectoryInfo**, which enables you to use each of the instance properties for each subdirectory. The following code example shows how to get all of the sub directories in the C:\fourthCoffee\tempData directory.

```
string directoryPath = @"C:\fourthCoffee\tempData";
DirectoryInfo directory = new DirectoryInfo(directoryPath);
```

```
DirectoryInfo[] subDirectories = directory.GetDirectories();
```

- The **GetFiles** method enables you to get a list of all the files within a specific directory on the file system. In contrast to the static **File.GetFiles** method, this instance method returns an array of type **FileInfo**, which enables you to use each of the instance properties for each file. The following example shows how to get all of the files in the C:\fourthCoffee\tempData directory.


```
string directoryPath = @"C:\fourthCoffee\tempData";
DirectoryInfo directory = new DirectoryInfo(directoryPath);
FileInfo[] subFiles = directory.GetFiles();
```

Depending on whether you require a simple one-line-of-code approach to manipulate a directory, or something that offers slightly more flexibility, either the static **Directory** or instance **DirectoryInfo** class should fulfill your requirements.

Manipulating File and Directory Paths

The **Path** class encapsulates file system utility functions

```
string settingsPath  = "..could be anything here..";

// Check to see if path has an extension.
bool hasExtension = Path.HasExtension(settingsPath);

...

// Get the extension from the path.
string pathExt = Path.GetExtension(settingsPath);

...

// Get path to temp file.
string tempPath = Path.GetTempFileName();
// Returns C:\Users\LeonidsP\AppData\Local\Temp\ABC.tmp
```

All files and all directories have a name, which when combined to point to a file in a directory, constitute a path. Different file systems can have different conventions and rules for what constitutes a path. The .NET Framework provides the **Path** class, which encapsulates a variety of file system utility functions that you can use to parse and construct valid file names, directory names, and paths within the Windows file system. These functions can be useful if your application needs to write a file to a temporary location, extract an element from a file system path, or even generate a random file name.

The following code shows how to create a new directory on the root of the C: drive.

Creating a Temporary Directory the Hard Way

```
string tempDirectoryPath = @"C:\fourthCoffee\tempData";
if (!Directory.Exists(tempDirectoryPath))
    Directory.CreateDirectory(tempDirectoryPath);
```

However, with the above approach, you are making many assumptions, including whether your application has the necessary privileges to perform I/O at the root of the C drive, and whether the C drive actually exists.

A better way is to use the static **GetTempPath** method provided by the **Path** class to get the path to the current user's Windows temporary directory.

Getting the Path to the Windows Temporary Directory

```
string tempDirectoryPath = Path.GetTempPath();
```

The **Path** class includes many other static methods that provide a good starting point for any custom I/O type functionality that your application may require. These methods include the following:

- The **HasExtension** method enables you to determine if the path your application

is processing has an extension. This provides a convenient way for you to determine if you are processing a file or a directory. The following example shows how to check whether the path has an extension.

```
string settingsPath  = "..could be anything here..";
bool hasExtension = Path.HasExtension(settingsPath);
```

- The **GetExtension** method enables you to get the extension from a file name. This method is particularly useful when you want to ascertain what type of file your application is processing. The following code example shows how to check whether the **settingsPath** variable contains a path that ends with the .txt extension.

```
string settingsPath  = "..could be anything here..";
string pathExt = Path.GetExtension(settingsPath);
if (pathExt == ".txt")
{
    // More processing here.
}
```

- The **GetTempFileName** enables you to create a new temp file in your local Windows temporary directory in a single transactional operation folder. This method then returns the absolute path to that file, ready for further processing. The following code shows how to invoke the **GetTempFileName** method.

```
string tempPath = Path.GetTempFileName();
// Returns C:\Users\LeonidsP\AppData\Local\Temp\ABC.tmp
```

> **Additional Reading:** For more information about **the** Path **class,** refer to the Path Class **page** at **https://aka.ms/moc-20483c-m6-pg1**.

Demonstration: Manipulating Files, Directories, and Paths

In this demonstration, you will use the **File**, **Directory**, and **Path** classes to build a utility that combines multiple files into a single file.

Demonstration steps

You will find the steps in the **Demonstration: Manipulating Files, Directories, and Paths** section on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD06_DEMO.md**.

Lesson 2: Serializing and Deserializing Data

Serialization is the process of converting data to a format that can be persisted or transported. Deserialization is the process of converting serialized data back to objects.

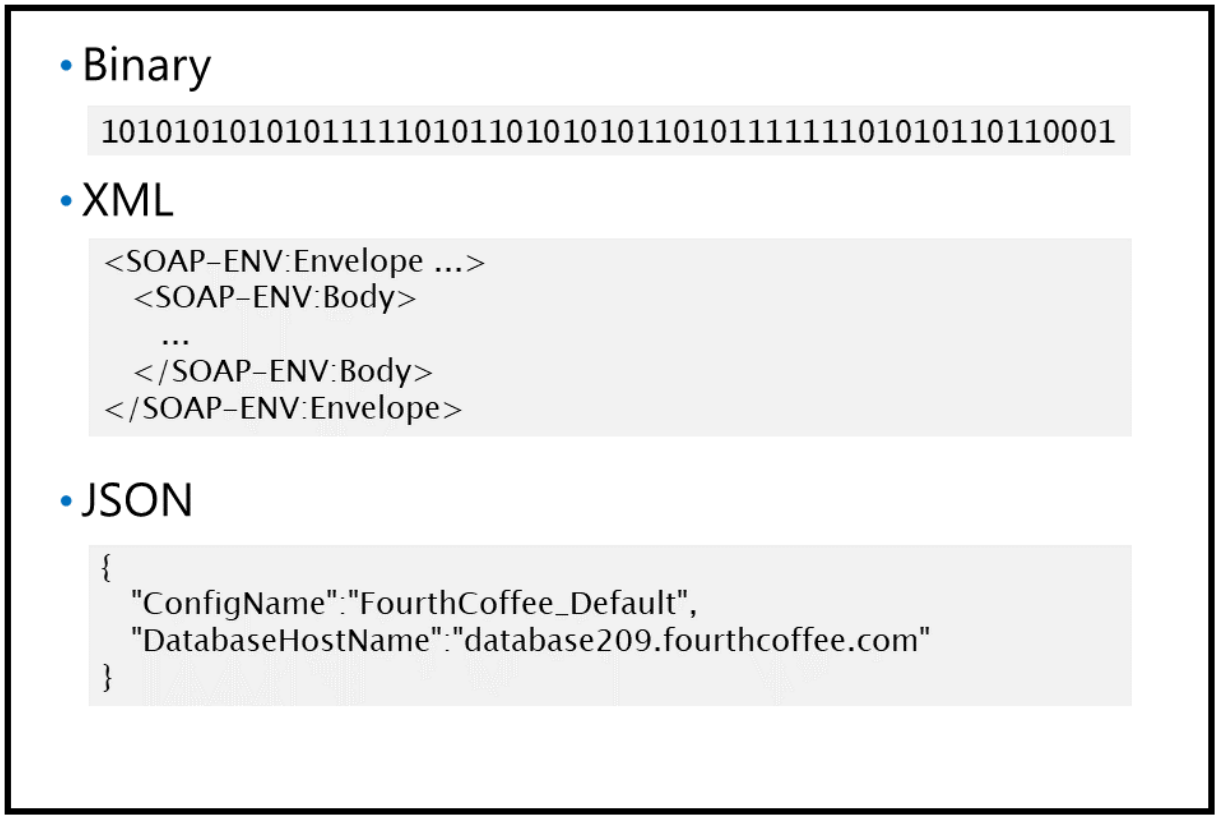
In this lesson, you will learn how to serialize objects to binary, XML, and JavaScript Object Notation (JSON), and how to create a custom serializer so that you can serialize objects into any format you choose.

Lesson objectives

After completing this lesson, you will be able to:

- Describe the purpose of serialization, and the formats that the .NET Framework supports.
- Create a custom type that is serializable.
- Serialize an object as binary.
- Serialize an object as XML.
- Serialize an object as JSON.
- Create a custom serializer by implementing the **IFormatter** interface.

What Is Serialization?



Applications typically process data. Data is read into memory, perhaps from a file or web service call, processed, and then passed to another component in the system for further processing. The components of a system may run on the same machine, but commonly components run on different platforms, on different hardware, and even in different geographical locations. The format of the data also needs to be lightweight so that it can be transported over a variety of protocols, such as HTTP or SOAP. Serialization is the process of converting the state of an object into a form that can be persisted or transported.

Serializable Formats

The requirements of your system, and how you intend to transport the data, will influence the serialization format you choose. The following table describes some of the common formats available.

Format	Description

Format	Description
Binary	Serializing data into the machine-readable binary format enables you to preserve the fidelity and state of an object between different instances of your application. Binary serialization is also fast and lightweight, because the binary format does not require the processing and storage of unnecessary formatting constructs. Binary serialization is commonly used when persisting and transporting objects between applications running on the same platform. Binary Example 1000000101110100011011100101011100011111110101110001100110010110111001111001011010100100001000011011000100000000011110000011100011000001000000110001001010100011101100101101000011100101000010110101010011111101110101100001101011001101110110100100001111010100111100000101101111111010111000110011001011011100111100101101010010000100001101100010000000001111000001110001100000100000011000100101010001110110010110100001110010100010110101010011111101110101000110000010000001100010010101000111011001011010000
XML	Serializing data into the XML format enables you to utilize an open standard that can be processed by any application, regardless of platform. In contrast to binary, XML does not preserve type fidelity; it only lets you serialize public members that your type exposes. The XML format is more verbose than binary, as the serialized data is formatted with XML constructs. This makes the XML format less efficient and more processor intensive during the serializing, deserializing, and transporting processes. However, the nature of XML as a plain text and free-form language allows messages to be transmitted across different applications and platforms. So long as the transmitter and receiver have agreed on a known contract, both can send and receive messages and convert them to the appropriate model within their respective environments. XML serialization was commonly used to serialize data that can be transported via the SOAP protocol to and from web services. However, due to SOAP's verbose and strict nature, this protocol has fallen out of favor, and is generally found today only in legacy environments. SOAP XML Example <SOAP-ENV:Envelope

Format	Description
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP- ENV=http://schemas.xmlsoap.org/soap/envelope/ xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"> <SOAP-ENV:Body> <a1:ServiceConfiguration id="ref-1" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/ FourthCoffeeSerializer/%2C%20Version%3D1.0.0.0%2C%20 Culture%3Dneutral%2C%20PublicKeyToken%3Dnull"> <ConfigName id="ref-3"> FourthCoffee_Default

Format	Description
	</ConfigName> <DatabaseHostName id="ref-4"> database209.fourthcoffee.com </DatabaseHostName> <ApplicationDataPath id="ref-5"> C:\fourthcoffee\applicationdata\ </ApplicationDataPath> </a1:ServiceConfiguration> </SOAP-ENV:Body> </SOAP-ENV:Envelope>

Format	Description

JSON	Serializing data into the JSON format enables you to utilize a lightweight, data-interchange format that is based on a subset of the JavaScript programming language. JSON is a simple text format that is human readable and also easy to parse by machine, irrespective of platform.
	JSON shares XML strengths as plain text and free form, making it cross platform. However, unlike XML, it has a much more concise syntax, which makes it cheaper to transmit and human readable. These advantages made JSON the prevalent language to transmit data today, and the de-facto standard in today's industry.
	JSON is commonly used to transport data between everything from Asynchronous JavaScript and XML (AJAX) calls to messages between webservices, because unlike SOAP, you are not limited to just communicating within the same domain.
	
	JSON Example
	
	{
	"ConfigName":"FourthCoffee_Default",
	"DatabaseHostName":"database209.fourthcoffee.com",
	"ApplicationDataPath":"C:\\fourthcoffee\\applicationdata\\"
	}
	
	Alternatively, if you want to serialize your data to a format that the .NET Framework does not natively support, you can implement your own custom serializer class.

Creating a Serializable Type

Implement the **ISerializable** interface

```
[Serializable]
public class ServiceConfiguration : ISerializable
{
  ...
  public ServiceConfiguration(
    SerializationInfo info, StreamingContext ctxt)
  {
    ...
  }

  public void GetObjectData(
    SerializationInfo info, StreamingContext context)
  {
    ...
  }
}
```

The .NET Framework provides many classes that are serializable. If you want to create your own types that are serializable, you need to ensure that the type definition incudes the necessary configuration and functionality for the serilalizer to consume. The .NET Framework provides the **System** and **System.Runtime.Serialization** namespaces, which provide classes to enable serialization support.

To create a serializable type, perform the following steps:

1. Define a default constructor.

```
public class ServiceConfiguration
{
   public ServiceConfiguration()   {      ...    }
}
```

2. Decorate the class with the **Serializable** attribute provided in the **System** namespace.

```
            [Serializable]
public class ServiceConfiguration
{
   ...
}
```

3. Implement the **ISerializable** interface provided in the **System.Runtime.Serialization** namespace. The **GetObjectData** method enables you to extract the data from your object during the serialization process.

```
[Serializable]
public class ServiceConfiguration : ISerializable
{
   ...
   public void GetObjectData(SerializationInfo info,
StreamingContext context)   {      ...   }
}
```

4. Define a deserialization constructor, which accepts **SerializationInfo** and **StreamingContext** objects as parameters. This constructor enables you to rehydrate your object during the deserialization process.

```
[Serializable]
public class ServiceConfiguration : ISerializable
{
   ...
   public ServiceConfiguration(SerializationInfo info,
StreamingContext ctxt)   {      ...   }
}
```

5. Define the public members that you want to serialize. You can instruct the serializer to ignore private fields by decorating them with the **NonSerialized**

attribute.

```
        ...
        [NonSerialized]
private Guid _internalId;
public string ConfigName { get; set; }
public string DatabaseHostName { get; set; }
public string ApplicationDataPath { get; set; }

...
```

The following code example shows the complete **ServiceConfiguration** class, which is serializable by any of the .NET Framework **IFormatter** implementations.

Serializable Type

```
[Serializable]
public class ServiceConfiguration : ISerializable
{
    [NonSerialized]
    private Guid _internalId;
    public string ConfigName { get; set; }
    public string DatabaseHostName { get; set; }
    public string ApplicationDataPath { get; set; }

    public ServiceConfiguration()
    {
    }
    public ServiceConfiguration(SerializationInfo info,
StreamingContext ctxt)
    {
        this.ConfigName
```

```
            = info.GetValue("ConfigName", typeof(string)).ToString();
        this.DatabaseHostName
            = info.GetValue("DatabaseHostName",
typeof(string)).ToString();
        this.ApplicationDataPath
            = info.GetValue("ApplicationDataPath",
typeof(string)).ToString();
    }
    public void GetObjectData(SerializationInfo info,
StreamingContext context)
    {
        info.AddValue("ConfigName", this.ConfigName);
        info.AddValue("DatabaseHostName", this.DatabaseHostName);
        info.AddValue("ApplicationDataPath",
this.ApplicationDataPath);
    }
}
```


Serializing Objects as Binary

- Serialize as binary

```
ServiceConfiguration config = ServiceConfiguration.Default;
IFormatter formatter = new BinaryFormatter();
FileStream buffer = File.Create("C:\\fourthcoffee\\config.txt");
formatter.Serialize(buffer, config);
buffer.Close();
```

- Deserialize from binary

```
IFormatter formatter = new BinaryFormatter();
FileStream buffer = File.OpenRead("C:\\fourthcoffee\\config.txt");
ServiceConfiguration config
   = formatter.Deserialize(buffer) as ServiceConfiguration;
buffer.Close();
```

The .NET Framework provides the **BinaryFormatter** class in the **System.Runtime.Serialization.Formatters.Binary** namespace, which you can use to serialize and deserialize objects as binary.

> **Note:** The **BinaryFormatter** and **SoapFormatter** classes implement the **IFormatter** interface. You can also implement the **IFormatter** interface to create your own custom serializer.

Serialize an Object by Using the BinaryFormatter Class

To serialize an object by using the **BinaryFormatter** class, perform the following steps:

1. Obtain a reference to the object you want to serialize.
2. Create an instance of the **BinaryFormatter** that you want to use to serialize your type.
3. Create a stream that you will use as a buffer to store the serialized data.
4. Invoke the **BinaryFormatter.Serialize** method, passing in stream that the serialized data will be buffered to, and the object you want to serialize.

The following code example shows how to use the **BinaryFormatter** class to serialize an object as binary.

BinaryFormatter Serialize Example

```
// Create the object you want to serialize.
ServiceConfiguration config = ServiceConfiguration.Default;
// Create the formatter you want to use to serialize the object.
IFormatter formatter = new BinaryFormatter();
// Create the stream that the serialized data will be buffered to.
FileStream buffer = File.Create(@"C:\fourthcoffee\config.txt");
// Invoke the Serialize method.
```

```
formatter.Serialize(buffer, config);
// Close the stream.
buffer.Close();
```

This example, serializes the **ServiceConfiguration** object, and writes the serialized data to a file. It is important to note that serialization doesn't just imply writing data to a file. Serialization is the process of transforming a type into another format, which you can then write to a file or database, or send over HTTP to a web service.

Deserialize an Object by Using the BinaryFormatter Class

Deserializing is the process of transforming your serialized object back into a format that your application can process. To deserialize an object by using the **BinaryFormatter** class, perform the following steps:

1. Create an instance of the **BinaryFormatter** that you want to use to deserialize your type.
2. Create a stream to read the serialized data.
3. Invoke the **BinaryFormatter.Deserialize** method, passing in stream that contains the serialized data.
4. Cast the result of the **BinaryFormatter.Deserialize** method call into the type of object that you are expecting.

The following code example shows how to use the **BinaryFormatter** class to deserialize binary data to an object.

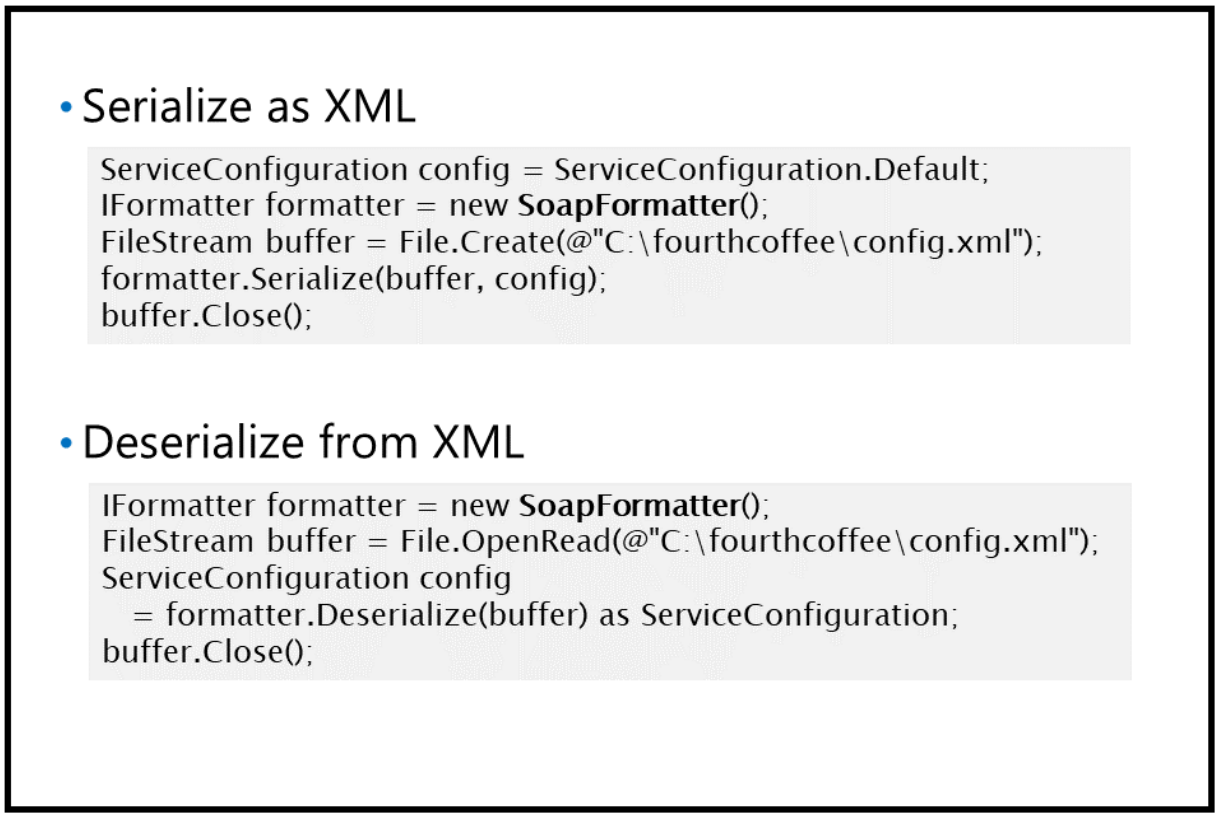
BinaryFormatter Deserialize Example

```
// Create the formatter you want to use to serialize the object.
IFormatter formatter = new BinaryFormatter();
// Create the stream that the serialized data will be buffered too.
FileStream buffer = File.OpenRead(@"C:\fourthcoffee\config.txt");
```

```
// Invoke the Deserialize method.
ServiceConfiguration config = formatter.Deserialize(buffer) as
ServiceConfiguration;
// Close the stream.
buffer.Close();
```

The above example reads the serialized binary data from a file, and then deserializes the binary into a **ServiceConfiguration** object. The process is the same for serializing and deserializing objects by using any formatters that implement the **IFormatter** interface. This includes the **SoapFormatter** class, and any custom formatters that you may implement.

Serializing Objects as XML



The .NET Framework provides the **SoapFormatter** class in the **System.Runtime.Serialization.Formatters.Soap** namespace, which you can use to serialize and deserialize objects as XML.

Serialize an Object by Using the SoapFormatter Class

The process for serializing data as XML is similar to the process of serializing to binary, with the exception that you use the **SoapFormatter** class.

The following code example shows how to use the **SoapFormatter** class to serialize an object as XML.

SoapFormatter Serialize Example


```
// Create the object you want to serialize.
ServiceConfiguration config = ServiceConfiguration.Default;
// Create the formatter you want to use to serialize the object.
IFormatter formatter = new SoapFormatter();
// Create the stream that the serialized data will be buffered too.
FileStream buffer = File.Create(@"C:\fourthcoffee\config.xml");
// Invoke the Serialize method.
formatter.Serialize(buffer, config);
// Close the stream.
buffer.Close();
```

Deserialize an Object by Using the SoapFormatter Class

The process for deserializing data from XML to an object is identical to the process of deserializing binary data, with the exception that you use the **SoapFormatter** class.

The following code example shows how to use the **SoapFormatter** class to deserialize XML data to an object.

SoapFormatter Deserialize Example


```
// Create the formatter you want to use to serialize the object.
IFormatter formatter = new SoapFormatter();
// Create the stream that the serialized data will be buffered too.
FileStream buffer = File.OpenRead(@"C:\fourthcoffee\config.xml");
// Invoke the Deserialize method.
```

```
ServiceConfiguration config = formatter.Deserialize(buffer) as
ServiceConfiguration;
// Close the stream.
buffer.Close();
```

Serializing Objects as JSON

- Serialize as JSON

```
ServiceConfiguration config = ServiceConfiguration.Default;
DataContractJsonSerializer jsonSerializer
  = new DataContractJsonSerializer(config.GetType());
FileStream buffer = File.Create(@"C:\fourthcoffee\config.txt");
jsonSerializer.WriteObject(buffer, config);
buffer.Close();
```

- Deserialize from JSON

```
DataContractJsonSerializer jsonSerializer = new
  DataContractJsonSerializer(
    typeof(ServiceConfiguration));
FileStream buffer = File.OpenRead(@"C:\fourthcoffee\config.txt");
ServiceConfiguration config = jsonSerializer.ReadObject(buffer)
  as ServiceConfiguration;
buffer.Close();
```

The .NET Framework also supports serializing objects as JSON by using the **DataContractJsonSerializer** class in the **System.Runtime.Serialization.Json** namespace. The JSON serialization steps are different because the **DataContractJsonSerializer** class is derived from the abstract **XmlObjectSerializer** class, and it is not an implementation of the **IFormatter** interface.

Serialize an Object by Using the DataContractJsonSerializer Class

To serialize an object by using the **DataContractJsonSerializer** class, perform the following steps:

1. Obtain a reference to the object that you want to serialize.
2. Create an instance of the **DataContractJsonSerializer** class that you want to

use to serialize your type. The constructor also requires you to pass in a **Type** object, representing the type of object you want to serialize.

3. Create a stream that you will use as a buffer to store the serialized data.

4. Invoke the **DataContractJsonSerializer.WriteObject** method, passing in stream that the serialized data will be buffered too, and the object you want to serialize.

The following code example shows how to use the **DataContractJsonSerializer** class to serialize an object as JSON.

DataContractJsonSerializer Serialize Example

```
// Create the object you want to serialize.
ServiceConfiguration config = ServiceConfiguration.Default;
// Create a DataContractJsonSerializer object that you will use to
serialize the
// object to JSON.
DataContractJsonSerializer jsonSerializer
   = new DataContractJsonSerializer(config.GetType());
// Create the stream that the serialized data will be buffered too.
FileStream buffer = File.Create(@"C:\fourthcoffee\config.txt");
// Invoke the WriteObject method.
jsonSerializer.WriteObject(buffer, config);
// Close the stream.
buffer.Close();
```

Deserialize an Object by using the DataContractJsonSerializer Class

To deserialize JSON to an object by using the **DataContractJsonSerializer** class, perform the following steps:

1. Create an instance of the **DataContractJsonSerializer** class that you want to use to serialize your type. The constructor also requires you to pass in a **Type**

object, representing the type of object you want to deserialize.

2. Create a stream that will read the serialized JSON into memory.

3. Invoke the **DataContractJsonSerializer.ReadObject** method, passing in the stream that contains the serialized data.

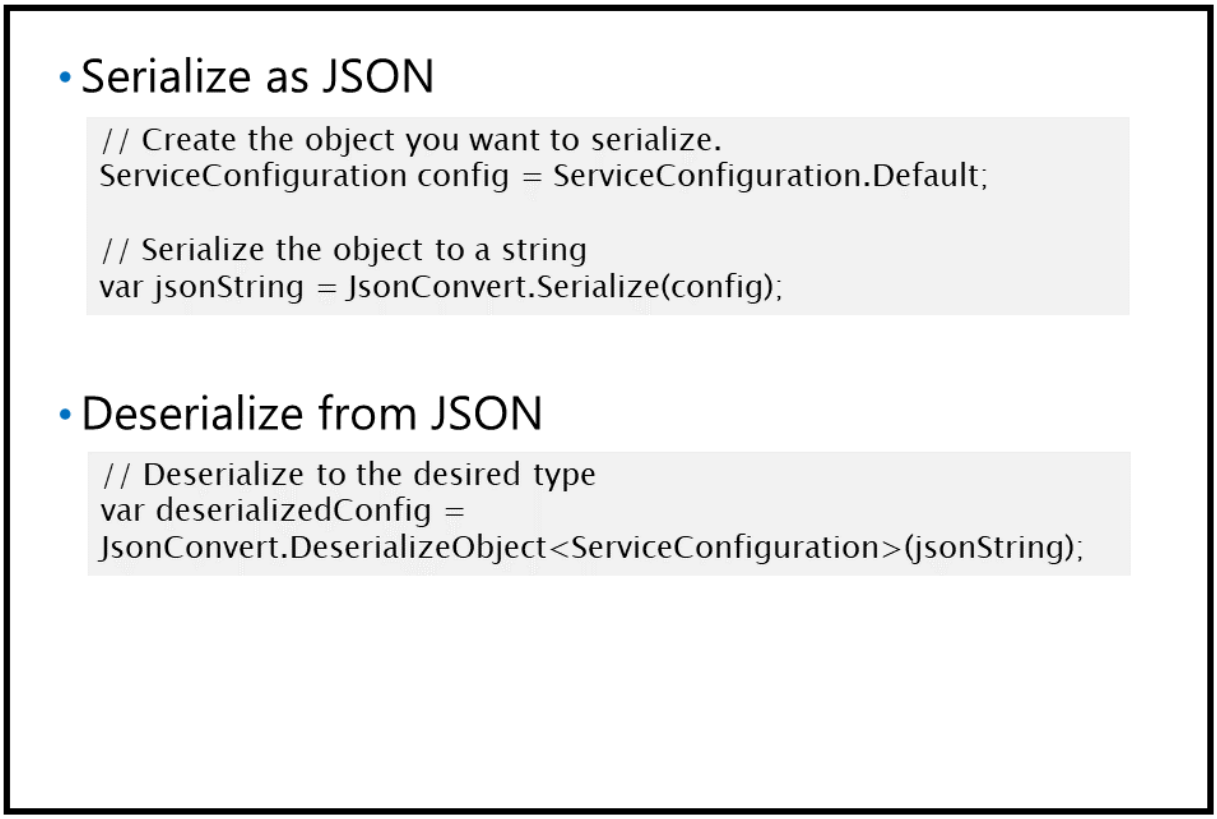
4. Cast the result of the **DataContractJsonSerializer.ReadObject** method call into the type of object you are expecting.

The following code example shows how to use the **DataContractJsonSerializer** class to deserialize JSON data to an object.

DataContractJsonSerializer Deserialize Example

```
// Create a DataContractJsonSerializer object that you will use to
// deserialize the JSON.
DataContractJsonSerializer jsonSerializer
   = new DataContractJsonSerializer(typeof(ServiceConfiguration));
// Create a stream that will read the serialized data.
FileStream buffer = File.OpenRead(@"C:\fourthcoffee\config.txt");
// Invoke the ReadObject method.
ServiceConfiguration config = jsonSerializer.ReadObject(buffer) as
ServiceConfiguration;
// Close the stream.
buffer.Close();
```


Serializing Objects as JSON by Using JSON.Net



While the .NET framework provides the built in **DataContractJsonSerializer** class to serialize objects to JSON, today's standard of using JSON in the .NET environment is Newtonsoft's **Json.NET** library. **Json.NET** provides an expansive library to create and query JSON. Most important for our purposes is their **JsonConvert** class that can serialize and deserialize classes to and from JSON.

To use the **Json.Net** library, you'll need to add it to your project by using **Nuget**, which is a package manager for .Net, allowing you to easily download and manage third party libraries in your project.

1. In **Solution Explorer**, right-click the project.
2. Select **Manage Nuget Packages**.
3. Select the **Browse** tab and search for **JSON**.
4. Select the **Newtonsoft.Json** package, and then click **Install**.

Unlike the .NET implementations of the **IFormatter** interface, **JsonConvert** doesn't require the data class to be decorated with the **Serializable** attribute. Any class can be converted to JSON without special treatment. However, **Json.NET** does provide a

set of attributes to specify exactly how the model will be serialized, if necessary. The most useful of them is perhaps the **JsonProperty** attribute, which allows to determine the name of the property in the serialized JSON string. Unlike Visual C#, JSON's naming standard for properties can vary, and usually follow the lowerCamelCase, snake_case, or even the kebab-case convention. This feature can be very useful, especially when communicating with other applications.

A normal data object ready to be used with **JsonConvert** can look something like this:

The following code example shows the complete **ServiceConfiguration** class, ready to be serialized with **Json.Net**.

JSON Serializable Type

```
public class ServiceConfiguration
{
    // JsonConvert ignores private members by default
    private Guid _internalId;
    // Map the properties with json naming conventions
    [JsonProperty("configName")]
    public string ConfigName { get; set; }
    [JsonProperty("databaseHostName")]
    public string DatabaseHostName { get; set; }
    [JsonProperty("applicationDataPath")]
    public string ApplicationDataPath { get; set; }
}
```


Serialize an Object by Using the JsonConvert Class

To serialize an object by using the **JsonConvert** class, perform the following steps:

1. Obtain a reference to the object that you want to serialize.
2. Invoke the **JsonConvert.Serialize** method. The method will automatically infer

the type processed, and serialize it accordingly.

The following code example shows how to use the **JsonConvert** class to serialize an object as JSON.

JsonConvert Serialize Example

```
// Create the object you want to serialize.
ServiceConfiguration config = ServiceConfiguration.Default;
var jsonString = JsonConvert.Serialize(config);
```

Deserialize an Object by Using the JsonConvert Class

To deserialize JSON to an object by using the **JsonConvert** class, perform the following steps:

1. Obtain the JSON string you need to deserialize.
2. Invoke the **JsonConvert.Desrialize<T>** method, passing in the string to be deserialized, as well as the target type as a generic type parameter.

The following code example shows how to use the **JsonConvert** class to deserialize JSON data to an object.

JsonConvert Deserialize Example

```
// Get the JSON string – Here we're assuming it's the same from the
previous example.
// Deserialize to the desired type
var deserializedConfig =
JsonConvert.DeserializeObject<ServiceConfiguration>(jsonString);
```

You might have noticed that unlike the previous topics, here we're serializing directly to a string, and not to a stream, and then to a file. You will find that most messages passed along are small enough not to warrant the use of a stream and saving them to a string is perfectly acceptable. Modern computers are powerful enough to handle quite large strings.

However, **Json.Net** allows serializing to and from streams very similarly to the internal .Net types, while still allowing to discard the **ISerializable** interface, and keeping **Json.Net's** lax usage of its attributes.

Serialize an Object by Using the JsonSerializer Class

To serialize an object by using the **JsonSerializer** class, perform the following steps:

1. Obtain a reference to the object that you want to serialize.
2. Create an instance of the **JsonSerializer** class that you want to use to serialize your type.
3. Create a stream writer to write the object.
4. Invoke the **JsonSerializer.Serialize** method, passing in the stream writer that the serialized data will be buffered too, and the object you want to serialize. The method will automatically infer the type processed and will serialize it accordingly.

The following code example shows how to use the **JsonSerializer** class to serialize an object as JSON and save it to a file.

JsonSerializer Serialize Example

```
// Create the serializer
var serializer = new JsonSerializer();
// Open the stream to the file
var fileWriter = File.CreateText(@"C:\fourthcoffee\config.json");
// Serialize and write the object to the file
```

```
serializer.Serialize(fileWriter, config);
// Close the stream
fileWriter.Close();
fileWriter.Dispose();
```

Deserialize an Object by Using the JsonSerializer Class

To deserialize JSON to an object by using the **JsonSerializer** class, perform the following steps:

1. Create an instance of the **JsonSerializer** class that you want to use to serialize your type.
2. Create a stream, stream reader, and **JsonTextReader** that will read the serialized JSON into the memory.
3. Invoke the **JsonSerializer.Deserialize<T>** method, passing in the **JsonTextReader** that contains the serialized data, as well as the target type as a generic type parameter.
4. Close all the readers and stream.

The following code example shows how to use the **JsonSerializer** class to deserialize JSON data to an object.

JsonSerializer Deserialize Example

```
// Create the serializer
var serializer = new JsonSerializer();
// Open a stream to the file
var fileReader = File.OpenRead(@"C:\fourthcoffee\config.json");
// Create a stream and json text readers
var textReader = new StreamReader(fileReader);
var jsonReader = new JsonTextReader(textReader);
// Deserialize to the desired type
```

```
var deserializedConfig =
serializer.Deserialize<ServiceConfiguration>(jsonReader);
// Close all the readers and the stream
jsonReader.Close();
textReader.Close();
textReader.Dispose();
fileReader.Close();
fileReader.Dispose();
```

> **Additional Reading:** For more information about the Json.Net library, you can turn to the following resources:
> Json.Net home page: **https://aka.ms/moc-20483c-m6-pg2**
> Json.Net Documentation: **https://aka.ms/moc-20483c-m6-pg3**
> JsonConvert class: **https://aka.ms/moc-20483c-m6-pg4**
> JsonPropertyAttribute class: **https://aka.ms/moc-20483c-m6-pg5**
> JsonSerializer class: **https://aka.ms/moc-20483c-m6-pg6**

Demonstration: Serializing Objects as JSON using JSON.Net

In this demonstration, you will see how to serialize and deserialize objects using JSON.NET.

Demonstration steps

You will find the steps in the **Demonstration: Serializing Objects as JSON using JSON.Net** section on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD06_DEMO.md**.

Creating a Custom Serializer

Implement the **IFormatter** interface

```
class IniFormatter : IFormatter
{
   public ISurrogateSelector SurrogateSelector { get; set; }
   public SerializationBinder Binder { get; set; }
   public StreamingContext Context { get; set; }

   public object Deserialize(Stream serializationStream)
   {
      ...
   }

   public void Serialize(Stream serializationStream, object graph)
   {
     ...
   }
}
```

You may want to serialize data into a format other than binary, XML, or JSON. The .NET Framework provides the **IFormatter** interface in the **System.Runtime.Serialization** namespace, so you can create your own formatter. Your custom formatter will then follow the same pattern as the **BinaryFormatter** and **SoapFormatter** classes.

To create your own formatter, perform the following steps:

1. Create a class that implements the **IFormatter** interface.
2. Create implementations for the **SurrogateSelector**, **Binder**, and **Context** properties.
3. Create implementations for the **Deserialize** and **Serialize** methods.

The following code example shows a custom formatter that can serialize and deserialize objects to the .ini format.

Custom IniFormatter

```
using System;
```

```
using System;
using System.Collections.Generic;
using System.IO;

using System.Reflection;
using System.Runtime.Serialization;
namespace FourthCoffee.Serializer
{
   class IniFormatter : IFormatter
   {
      static readonly char[] _delim = new char[] { '=' };
      public ISurrogateSelector SurrogateSelector { get; set; }
      public SerializationBinder Binder { get; set; }
      public StreamingContext Context { get; set; }
      public IniFormatter()
      {
         this.Context
            = new StreamingContext(StreamingContextStates.All);
      }
      public object Deserialize(Stream serializationStream)
      {
         StreamReader buffer
            = new StreamReader(serializationStream);
         // Get the type from the serialized data.
         Type typeToDeserialize = this.GetType(buffer);
          // Create default instance of object using type name.
         Object obj
            =
FormatterServices.GetUninitializedObject(typeToDeserialize);
         // Get all the members for the type.
         MemberInfo[] members
            =
FormatterServices.GetSerializableMembers(obj.GetType(),
this.Context);
         // Create dictionary to store the variable names and any
serialized data.
         Dictionary<string, object> serializedMemberData
            = new Dictionary<string, object>();
         // Read the serialized data, and extract the variable names
```

```
            // Read the serialized data, and extract the variable names
            // and values as strings.
            while (buffer.Peek() >= 0)
            {
               string line = buffer.ReadLine();
               string[] sarr = line.Split(_delim);
               // key = variable name, value = variable value.
               serializedMemberData.Add(
                  sarr[0].Trim(),   // Variable name.
                  sarr[1].Trim());  // Variable value.
            }
            // Close the underlying stream.
            buffer.Close();
            // Create a list to store member values as their correct
    type.
            List<object> dataAsCorrectTypes = new List<object>();
            // For each of the members, get the serialized values as
    their correct type.
            for (int i = 0; i < members.Length; i++)
            {
               FieldInfo field = members[i] as FieldInfo;
               if(!serializedMemberData.ContainsKey(field.Name))
                  throw new SerializationException(field.Name);
                // Change the type of the value to the correct type
                // of the member.
                object valueAsCorrectType = Convert.ChangeType(
                   serializedMemberData[field.Name],
                   field.FieldType);
                dataAsCorrectTypes.Add(valueAsCorrectType);
            }
            // Populate the object with the deserialized values.
            return FormatterServices.PopulateObjectMembers(
               obj,
               members,
               dataAsCorrectTypes.ToArray());
         }
         public void Serialize(Stream serializationStream, object graph)
         {
```

```
        {
            // Get all the fields that you want to serialize.
            MemberInfo[] allMembers

                =
FormatterServices.GetSerializableMembers(graph.GetType(),
this.Context);
             // Get the field data.
            object[] fieldData = FormatterServices.GetObjectData(graph,
allMembers);
            // Create a buffer to write the serialized data too.
            StreamWriter sw = new StreamWriter(serializationStream);
            // Write the name of the class to the firstline.
            sw.WriteLine("@ClassName={0}", graph.GetType().FullName);
            // Iterate the field data.
            for (int i = 0; i < fieldData.Length; ++i)
            {
                 sw.WriteLine("{0}={1}",
                     allMembers[i].Name,        // Member name.
                     fieldData[i].ToString());  // Member value.
            }
            sw.Close();
        }
        private Type GetType(StreamReader buffer)
        {
             string firstLine = buffer.ReadLine();
             string[] sarr = firstLine.Split(_delim);
             string nameOfClass = sarr[1];
             return Type.GetType(nameOfClass);
         }
    }
}
```

Lesson 3: Performing I/O by Using Streams

When you work with data, whether the data is stored in a file on the file system or on a web server accessible over HTTP, the data sometimes becomes too large to load into memory and transmit in a single transactional operation. For example, imagine trying to load a 200-gigabyte video file from the file system into memory in a single operation. Not only would the operation take a long time, but it would also consume a large amount of memory.

In this lesson, you will learn how to use streams to read from and write to files without having to cache the entire file in memory.

Lesson objectives

After completing this lesson, you will be able to:

- Describe the purpose of a stream.
- Describe the different types of streams provided in the .NET Framework.
- Describe how to use a stream.

What are Streams?

- The **System.IO namespace** contains a number of stream classes, including:
  - The abstract **Stream** base class
  - The **FileStream** class
  - The **MemoryStream** class
- Typical stream operations include:
  - Reading chunks of data from a stream
  - Writing chunks of data to a stream
  - Querying the position of the stream

The .NET Framework enables you to use streams. A stream is a sequence of bytes, which could come from a file on the file system, a network connection, or memory. Streams enable you to read from or write to a data source in small, manageable data packets. Typically, streams provide the following operations:

- Reading chunks of data into a type, such as a byte array.
- Writing chunks of data from a type to a stream.
- Querying the current position in the stream and modifying a specific selection of bytes at the current position.

Streaming in the .NET Framework

The .NET Framework provides several stream classes that enable you to work with a variety of data and data sources. When choosing which stream classes to use, you need to consider the following:

- What type of data you are reading or writing, for example, binary or alphanumeric.
- Where the data is stored, for example, on the local file system, in memory, or on a web server over a network.

The .NET Framework class library provides several classes in the **System.IO** namespace that you can use to read and write files by using streams. At the highest level of abstraction, the **Stream** class defines the common functionality that all streams provide. The class provides a generic view of a sequence of bytes, together with the operations and properties that all streams provide. Internally, a **Stream** object maintains a pointer that refers to the current location in the data source. When you first construct a **Stream** object over a data source, this pointer is positioned immediately before the first byte. As you read and write data, the **Stream** class advances this pointer to the end of the data that is read or written.

You cannot use the **Stream** class directly. Instead, you instantiate specializations of this class that are optimized to perform stream-based I/O for specific types of data sources. For example, the **FileStream** class implements a stream that uses a disk

file as the data source, and the **MemoryStream** class implements a stream that uses a block of memory as the data source.

Types of Streams in the .NET Framework

- Classes that enable access to data sources include:

Class	Description
FileStream	Exposes a stream to a file on the file system.
MemoryStream	Exposes a stream to a memory location.
NetworkStream	Exposes a stream to a network location.

- Classes that enable reading and writing to and from data source streams include:

Class	Description
StreamReader	Read textual data from a source stream.
StreamWriter	Write textual data to a source stream.
BinaryReader	Read binary data from a source stream.
BinaryWriter	Write binary data to a source stream.

The .NET Framework provides many stream classes that you can use to read and write different types of data from different types of data sources. The following table describes some of these stream classes, and when you might want to use them.

Stream class	Description
FileStream	Enables you to establish a stream to a file on the file system. The **FileStream** class handles operations such as opening and closing the file, and provides access to the file through a raw sequence of bytes.
MemoryStream	Enables you to establish a stream to a location in memory. The **MemoryStream** class handles operations such as acquiring the in-memory storage, and provides access to the memory location through a raw sequence of bytes.
NetworkStream	Enables you to establish a stream to a network location in memory. The **NetworkStream** class handles operations such as opening and closing a connection to the network location, and provides access to the network location through a raw sequence of bytes.

A stream that is established by using a **FileStream**, **MemoryStream**, or **NetworkStream** object is just a raw sequence of bytes. If the source data is structured, you must convert the byte sequence into the appropriate types. This can be a time-consuming and error-prone task. However, the .NET Framework contains classes that you can use to read and write textual data and primitive types in a stream that you have opened by using the **FileStream**, **MemoryStream**, or **NetworkStream** classes. The following table describes some of the stream reader and writer classes.

Stream class	Description
StreamReader	Enables you to read textual data from an underlying data source stream, such as a **FileStream**, **MemoryStream**, or **NetworkStream** object.
StreamWriter	Enables you to write textual data to an underlying data source stream, such as a **FileStream**, **MemoryStream**, or **NetworkStream** object.
BinaryReader	Enables you to read binary data from an underlying data source stream, such as a **FileStream**, **MemoryStream**, or **NetworkStream** object.
BinaryWriter	Enables you to write binary data to an underlying data source stream, such as a **FileStream**, **MemoryStream**, or **NetworkStream** object.

Reading and Writing Binary Data by Using Streams

You can use the **BinaryReader** and **BinaryWriter** classes to stream binary data

```
string filePath = "C:\\fourthcoffee\\applicationdata\\settings.txt";

// Underlying stream to file on the file system.
FileStream file = new FileStream(filePath);

// BinaryReader object exposes read operations on the underlying
// FileStream object.
BinaryReader reader = new BinaryReader(file);

// BinaryWriter object exposes write operations on the underlying
// FileStream object.
BinaryWriter writer = new BinaryWriter(file);
```

Many applications store data in raw binary form for a number of reasons, such as the following:

- Writing binary data is fast.
- Binary data takes up less space on disk.
- Binary data is not human readable.

You can read and write data in a binary format in your .NET Framework applications by using the **BinaryReader** and **BinaryWriter** classes.

To read or write binary data, you construct a **BinaryReader** or **BinaryWriter** object by providing a stream that is connected to the source of the data that you want to read or write, such as a **FileStream** or **MemoryStream** object.

The following code example shows how to initialize the **BinaryReader** and **BinaryWriter** classes, passing a **FileStream** object.

Initializing a BinaryReader and BinaryWriter Object

```
string filePath = @"C:\fourthcoffee\applicationdata\settings.txt";
FileStream file = new FileStream(filePath);
...
BinaryReader reader = new BinaryReader(file);
...
BinaryWriter writer = new BinaryWriter(file);
```

After you have created a **BinaryReader** object, you can use its members to read the binary data. The following list describes some of the key members:

- The **BaseStream** property enables you to access the underlying stream that the **BinaryReader** object uses.
- The **Close** method enables you to close the **BinaryReader** object and the

underlying stream.

- The **Read** method enables you to read the number of remaining bytes in the stream from a specific index.
- The **ReadByte** method enables you to read the next byte from the stream, and advance the stream to the next byte.
- The **ReadBytes** method enables you to read a specified number of bytes into a byte array.

  Similarly, the **BinaryWriter** object exposes various members to enable you to write data to an underlying stream. The following list describes some of the key members.

- The **BaseStream** property enables you to access the underlying stream that the **BinaryWiter** object uses.
- The **Close** method enables you to close the **BinaryWiter** object and the underlying stream.
- The **Flush** method enables you to explicitly flush any data in the current buffer to the underlying stream.
- The **Seek** method enables you to set your position in the current stream, thus writing to a specific byte.
- The **Write** method enables you to write your data to the stream, and advance the stream. The **Write** method provides several overloads that enable you to write all primitive data types to a stream.

Reading Binary Data

The following code example shows how to use the **BinaryReader** and **FileStream** classes to read a file that contains a collection of bytes. This example uses the **Read** method to advance through the stream of bytes in the file.

BinaryReader Example

```
// Source file path.
string sourceFilePath =
    @"C:\fourthcoffee\applicationdata\settings.txt ";
// Create a FileStream object so that you can interact with the file
// system.
FileStream sourceFile = new FileStream(
    sourceFilePath,  // Pass in the source file path.
    FileMode.Open,   // Open an existing file.
    FileAccess.Read);// Read an existing file.
// Create a BinaryWriter object passing in the FileStream object.
BinaryReader reader = new BinaryReader(sourceFile);
// Store the current position of the stream.
int position = 0;
// Store the length of the stream.
int length = (int)reader.BaseStream.Length;
// Create an array to store each byte from the file.
byte[] dataCollection = new byte[length];
int returnedByte;
while ((returnedByte = reader.Read()) != -1)
{
    // Set the value at the next index.
    dataCollection[position] = (byte)returnedByte;
    // Advance our position variable.
    position += sizeof(byte);
}
// Close the streams to release any file handles.
reader.Close();
sourceFile.Close();
```

Writing Binary Data

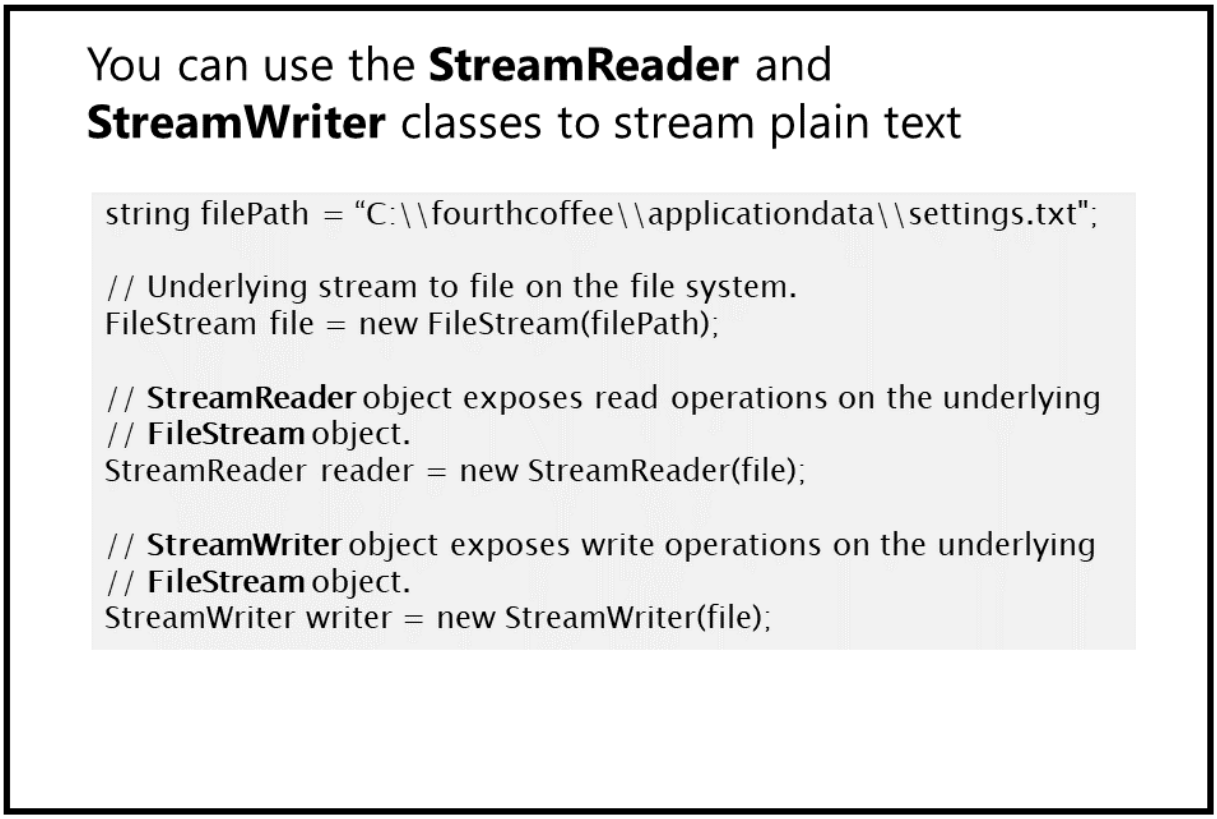
The following code example shows how to use the **BinaryWriter** and **FileStream** classes to write a collection of four byte integers to a file.

BinaryWriter Example

```
string destinationFilePath =
@"C:\fourthcoffee\applicationdata\settings.txt";
// Collection of bytes.
byte[] dataCollection = { 1, 4, 6, 7, 12, 33, 26, 98, 82, 101 };
// Create a FileStream object so that you can interact with the file
// system.
FileStream destFile = new FileStream(
    destinationFilePath, // Pass in the destination path.
    FileMode.Create,     // Always create new file.
    FileAccess.Write);   // Only perform writing.
// Create a BinaryWriter object passing in the FileStream object.
BinaryWriter writer = new BinaryWriter(destFile);
// Write each byte to stream.
foreach (byte data in dataCollection)
{
    writer.Write(data);
}
// Close both streams to flush the data to the file.
writer.Close();
destFile.Close();
```

Note: It's best to use the **IDisposable** interface with a **using statement**, as opposed to directly using the **Open** and **Close** methods. You can learn more about them in Module 11, Lesson 2, Topic 2 – "Implementing the Dispose Pattern".

Reading and Writing Text Data by Using Streams



In addition to storing data as raw binary data, you can also store data as plain text. You may want to do this in your application if the persisted data needs to be human readable.

The process for reading from and writing plain text to a file is very similar to reading and writing binary data, except that you use the **StreamReader** and **StreamWriter** classes.

When you initialize the **StreamReader** or **StreamWriter** classes, you must provide a stream object to handle the interaction with the data source.

The following code example shows how to initialize the **StreamReader** and **StreamWriter** classes, passing a **FileStream** object.

Initializing a StreamReader and StreamWriter Object

```
string destinationFilePath =
@"C:\fourthcoffee\applicationdata\settings.txt";
FileStream file = new FileStream(destinationFilePath);
...
```

```
StreamReader reader = new StreamReader(file);
...
StreamWriter writer = new StreamWriter(file);
```

After you have created a **StreamReader** object, you can use its members to read the plain text. The following list describes some of the key members:

- The **Close** method enables you to close the **StreamReader** object and the underlying stream.
- The **EndOfStream** property enables you to determine whether you have reached the end of the stream.
- The **Peek** method enables you to get the next available character in the stream, but does not consume it.
- The **Read** method enables you to get and consume the next available character in the stream. This method returns an **int** variable that represents the binary of the character, which you may need to explicitly convert.
- The **ReadBlock** method enables you to read an entire block of characters from a specific index from the stream.
- The **ReadLine** method enables you to read an entire line of characters from the stream.
- The **ReadToEnd** method enables you to read all characters from the current position in the stream.

Similarly, the **StreamWriter** object exposes various members to enable you to write data to an underlying stream. The following list describes some of the key members:

- The **AutoFlush** property enables you to instruct the **StreamWriter** object to flush data to the underlying stream after every write call.
- The **Close** method enables you to close the **StreamWriter** object and the underlying stream.

- The **Flush** method enables you to explicitly flush any data in the current buffer to the underlying stream.
- The **NewLine** property enables you to get or set the characters that are used for new line breaks.
- The **Write** method enables you to write your data to the stream, and to advance the stream.
- The **WriteLine** method enables you to write your data to the stream followed by a new line break, and then advance the stream.

These members provide many options to suit many different requirements. If you do not want to store the entire file in memory in a single chunk, you can use a combination of the **Peek** and **Read** methods to read each character, one at a time. Similarly, if you want to write lines of text to a file one at time, you can use the **WriteLine** method.

Reading Plain Text

The following code example shows how to use the **StreamReader** and **FileStream** classes to read a file that contains plain text. This example uses the **Peak** method to advance through the stream of characters in the file.

StreamReader Example


```
string sourceFilePath =
    @"C:\fourthcoffee\applicationdata\settings.txt ";
// Create a FileStream object so that you can interact with the file
// system.
FileStream sourceFile = new FileStream(
    sourceFilePath,  // Pass in the source file path.
    FileMode.Open,   // Open an existing file.
    FileAccess.Read);// Read an existing file.
StreamReader reader = new StreamReader(sourceFile);
StringBuilder fileContents = new StringBuilder();
```

```
// Check to see if the end of the file
// has been reached.
while (reader.Peek() != -1)
{
    // Read the next character.
    fileContents.Append((char)reader.Read());
}
// Store the file contents in a new string variable.
string data = fileContents.ToString();
// Always close the underlying streams release any file handles.
reader.Close();
sourceFile.Close();
```

Writing Plain Text

The following code example shows how to use the **StreamWriter** and **FileStream** classes to write a string to a new file on the file system.

StreamWriter Example

```
string destinationFilePath =
    @"C:\fourthcoffee\applicationdata\settings.txt ";
string data = "Hello, this will be written in plain text";
// Create a FileStream object so that you can interact with the file
// system.
FileStream destFile = new FileStream(
    destinationFilePath, // Pass in the destination path.
    FileMode.Create,     // Always create new file.
    FileAccess.Write);   // Only perform writing.
// Create a new StreamWriter object.
StreamWriter writer = new StreamWriter(destFile);
// Write the string to the file.
writer.WriteLine(data);
// Always close the underlying streams to flush the data to the file
// and release any file handles.
```

```
writer.Close();
destFile.Close();
```

Demonstration: Generating the Grades Report Lab

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Demonstration steps

You will find the steps in the **Demonstration: Generating the Grades Report Lab** section on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD06_DEMO.md**.

Lab: Generating the Grades Report

Scenario

You have been asked to upgrade the Grades Prototype application to enable users to save a student's grades as an XML file on the local disk. The user should be able to click a new button on the StudentProfile view that asks the user where they would like to save the file, displays a preview of the data to the user, and asks the user to confirm that they wish to save the file to disk. If they do, the application should save the grade data in XML format in the location that the user specified.

Objectives

After completing this lab, you will be able to:

- Serialize data to a memory stream.
- Deserialize data from a memory stream.
- Save serialized data to a file.

Lab setup

Estimated Time: 60 minutes

You will find the high-level steps on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD06_LAB_MANUAL.md**.

You will find the detailed steps on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD06_LAK.md**.

Exercise 1: Serializing Data for the Grades Report as XML

Scenario

In this exercise, you will write code that runs when the user clicks the **Save Report** button on the **Student Profile** view. You will enable a user to specify where to save the Grade Report, and to serialize the grades data so it is ready to save to a file.

You will use the **SaveFileDialog** object to ask the user for the file name and location where they want to save the file. You will extract the grade data from the application data source and store it in a list of Grade objects.

You will then write the **FormatAsXMLStream** method. This method will use an **XmlWriter** object to create an XML document and populate it with grade information from the list of Grade objects. Finally, you will debug the application and view the data held in the memory stream.

> **Result**: After completing this exercise, users will be able to specify the location for the Grades Report file.

Exercise 2: Previewing the Grades Report

Scenario

In this exercise, you will write code to display a preview of the report to the user before saving it.

First, you will add code to the **SaveReport_Click** method to display the XML document to the user in a message box. To display the document, you need to build a string representation of the XML document that is stored in the **MemoryStream** object. Finally, you will verify that your code functions as expected by running the application and previewing the contents of a report.

Result: After completing this exercise, users will be able to preview a report before saving it.

Exercise 3: Persisting the Serialized Grade Data to a File

Scenario

In this exercise, you will write the grade data to a file on the local disk.

You will begin by modifying the existing preview dialog box to ask the user if they wish to save the file. If they wish to save the file, you will use a **FileStream** object to copy the data from the **MemoryStream** to a physical file. Then you will run the application, generate and save a report, and then verify that the report has been saved in the correct location in the correct format.

Result: After completing this exercise, users will be able to save student reports to the local hard disk in XML format.

Module review and takeaways

In this module, you have learned how to work with the file system by using a number of classes in the System.IO namespace, and how to serialize application data to different formats.

Review Question(s)

Check Your Knowledge

Discovery

You are a developer working on the Fourth Coffee Windows Presentation Foundation (WPF) client application. You have been asked to store some settings in a plain text file in the user's temporary folder on the file system. Briefly explain which classes and methods you could use to achieve this.
The Path.GetTempPath(), Directory.Exists(…), Directory.CreateDirectory(), File.WriteAllLines(…), and File.ReadAllLines(…) methods.

Show solution Reset

Check Your Knowledge

Discovery

You are a developer working for Fourth Coffee. A bug has been raised and you have been asked to investigate. To help reproduce the error, you have decided to add some logic to persist the state of the application to disk, when the application encounters the error. All the types in the application are serializable, and it would be advantageous if the persisted state was human readable. What approach will you take?
The JsonConvert class.

Show solution Reset

Check Your Knowledge

Select the best answer

You are a developer working for Fourth Coffee. You have been asked to write some code to process a 100 GB video file. Your code needs to transfer the file from one location on disk, to another location on disk, without reading the entire file into memory. Which classes would you use to read and write the file?

The MemoryStream, BinaryReader and BinaryWriter classes.

The FileStream, BinaryReader and BinaryWriter classes.

The BinaryReader and BinaryWriter classes.

The FileStream, StreamReader and StreamWriter classes.

The MemoryStream, StreamReader and StreamWriter classes.

Check answer Show solution Reset