Microsoft® Official Course

Module 6

Reading and Writing Local Data

Module Overview

- Reading and Writing Files
- Serializing and Deserializing Data
- Performing I/O by Using Streams

The Why?

- Read/Write data to the local file system (File):
  - E.g. a Windows service, which does not have a user interface, still requires the ability to log information for someone or some other system to consume.
- Manipulating file on the file system (File, FileInfo)
  - E.g. an application that copies a file to a temporary location on disk before performing additional processing. Or it may need to read some metadata associated with the file, such as the file creation time.
- Manipulating directories
  - E.g. applications 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.
- Serializing/Deserializing Data
  - “Ship” data over the network, or save/open data (objects)
- Streams
  - Reading and writing data in single atomic operations is acceptable when processing small files. However, when you are working with large amounts of data, such operations are inefficient and can consume too much memory and processor time.
    - E.g. suppose you have a 100GB of data to process?

Lesson 1: Reading and Writing Files

- Reading and Writing Data by Using the File Class
- Manipulating Files
- Manipulating Directories
- Manipulating File and Directory Paths
- Demonstration: Manipulating Files, Directories, and Paths

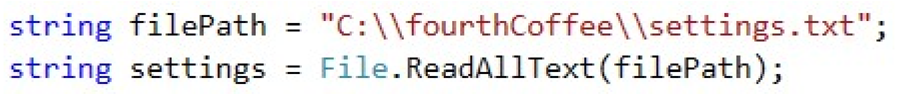
- The .NET Framework provides several classes that you can use to interact with files, directories, and paths. These classes include the following:
    - The **File** class.
    - The **FileInfo** class.
    - The **Directory** class.
    - The **DirectoryInfo** class.
    - The **Path** class.

Reading Data by Using the File Class

- The **System.IO namespace** contains classes for manipulating files and directories

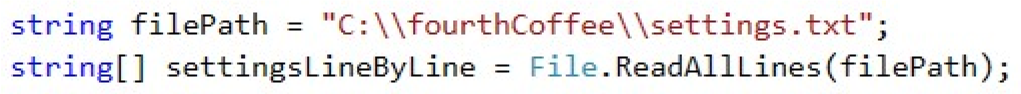
- The **File** class provides various static methods that enable you to perform atomic read and write operations. The methods are atomic because they wrap several low-level functions into a single, convenient method call.

- The **File** class contains atomic read methods, including:
    - 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
  - **ReadAllText(...)** - read the entire contents of a file into memory (a string)



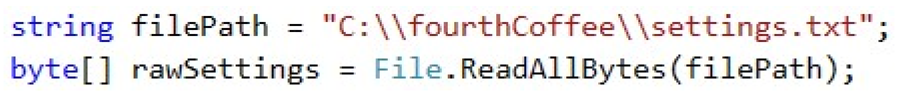
```
string filePath = "C:\\fourthCoffee\\settings.txt";
string settings = File.ReadAllText(filePath);
```

  - **ReadAllLines(...)** - read the contents of a file and store each line at a new index in a string array



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

  - **ReadAllBytes(...)** - read the contents of a file as binary data and store the data in a byte array.

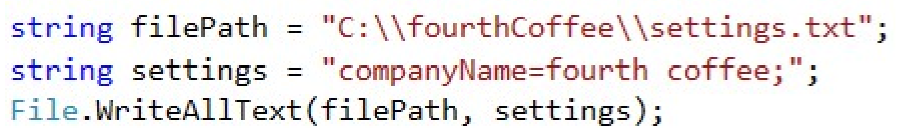


```
string filePath = "C:\\fourthCoffee\\settings.txt";
byte[] rawSettings = File.ReadAllBytes(filePath);
```

Writing Data by Using the File Class

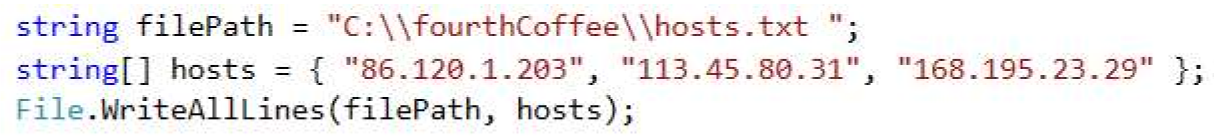
- The **File** class contains atomic write methods, including:
  - **Writexxx** methods create a new file with the new data.
    - If the file does exist, the Writexxx methods overwrite the existing file with the new data.
  - **Appendxxx** methods also create a new file with the new data.
    - However, if the file does exist, the new data is written to the end of the existing file.

- **WriteAllText(…)**
  - enables you to write the contents of a **string** variable to a file. If the file exists, its contents will be overwritten



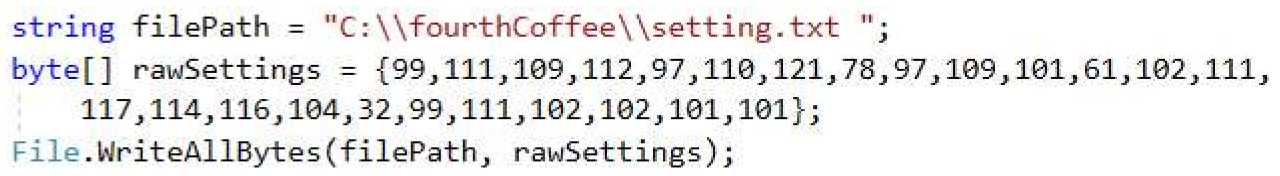
```
string filePath = "C:\\fourthCoffee\\settings.txt";
string settings = "companyName=fourth coffee;";
File.WriteAllText(filePath, settings);
```

- **WriteAllLines(…)**
  - 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.



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

- **WriteAllBytes(…)**
  - enables you to write the contents of a **byte array** to a binary file.



```
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);
```

Manipulating Files

- **File** class provides static members
  - **Copy** method - copy an existing file to a different directory on the file system.
    - Third parameter indicates that the copy process should overwrite an existing file if it exists at the destination path. If you pass **false** and the file already exists, the CLR will throw a **System.IO.IOException**.
  - **Delete** method enables you to delete an existing file from the file system
  - **Exists** method enables you to check whether a file exists on the file system
  - **GetCreationTime** method enables you to read the date time stamp that describes when a file was created, from the metadata associated with the file.

```
File.Copy(sourceSettingsPath, destinationSettingsPath, overWrite);
File.Delete(...);
bool exists = File.Exists(...);
DateTime createdOn = File.GetCreationTime(...);
```

- **FileInfo** class provides instance members
    - exposes metadata associated with the file through **properties**.
    - exposes operations through **methods**.
  - **CopyTo** method enables you to copy an existing file to a different directory on the file system.
    - Second parameter indicates that the copy process should overwrite an existing file if it exists at the destination path. If you pass **false** and the file already exists, the CLR will throw a **System.IO.IOException**.
  - **Delete** method enables you to delete a file.
  - **DirectoryName** property enables you to get the directory path to the file.
  - **Exists** method enables you to determine if the file exists within the file system
  - **Extension** property enables you to get the file extension of a file.
  - **Length** property enables you to get the length of the file in bytes.

```
FileInfo file = new FileInfo(...);
...
string name = file.DirectoryName;
bool exists = file.Exists;
file.Delete();
...
file.CopyTo(destSettingsPath, overWrite)
long length = file. Length
```

Manipulating Directories

- **Directory** class provides static members
  - **CreateDirectory** method enables you to create a new directory on the file system.
  - **Delete** method enables you to delete a directory at a specific path.
    - **recursDelSubContent** 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**.
  - **Exists** method enables you to determine if a directory exists on the file system
  - **GetDirectories** method enables you to get a list of all subdirectories within a specific directory on the file system.
  - **GetFiles** method enables you to get a list of all the files within a specific directory on the file system.

```
string dirPath = "C:\\fourthCoffee\\tempData";
Directory.CreateDirectory(dirPath);
Directory.Delete(dirPath, recursDelSubContent);
bool exists = Directory.Exists(...);
string[] files = Directory.GetFiles(...);
```

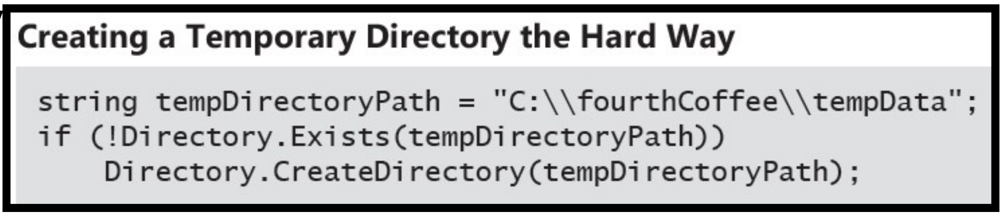
- **DirectoryInfo** class provides instance members (in-memory ...)
  - First must create an instance of the class
  - **Create** method enables you to create a new directory on the file system.
  - **Delete** method enables you to delete a directory at a specific path.
    - **recursDelSubContent** parameter ...
  - **Exists** property enables you to determine if a directory exists on the file system.
  - **FullName** property enables you to get the full path to the directory. GetFiles, GetDirectories

```
DirectoryInfo directory = new DirectoryInfo(...);
...
string path = directory.FullName;
bool exists = directory.Exists;
FileInfo[] files = directory.GetFiles();
```

Manipulating File and Directory Paths

Path class encapsulates file system utility functions

- by using these types of utility classes, you will write less code, save time, and be able to concentrate on more complex I/O functionality
- GetTempPath method (Path class) gets the path to the current user's Windows temporary directory



Creating a Temporary Directory the Hard Way

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

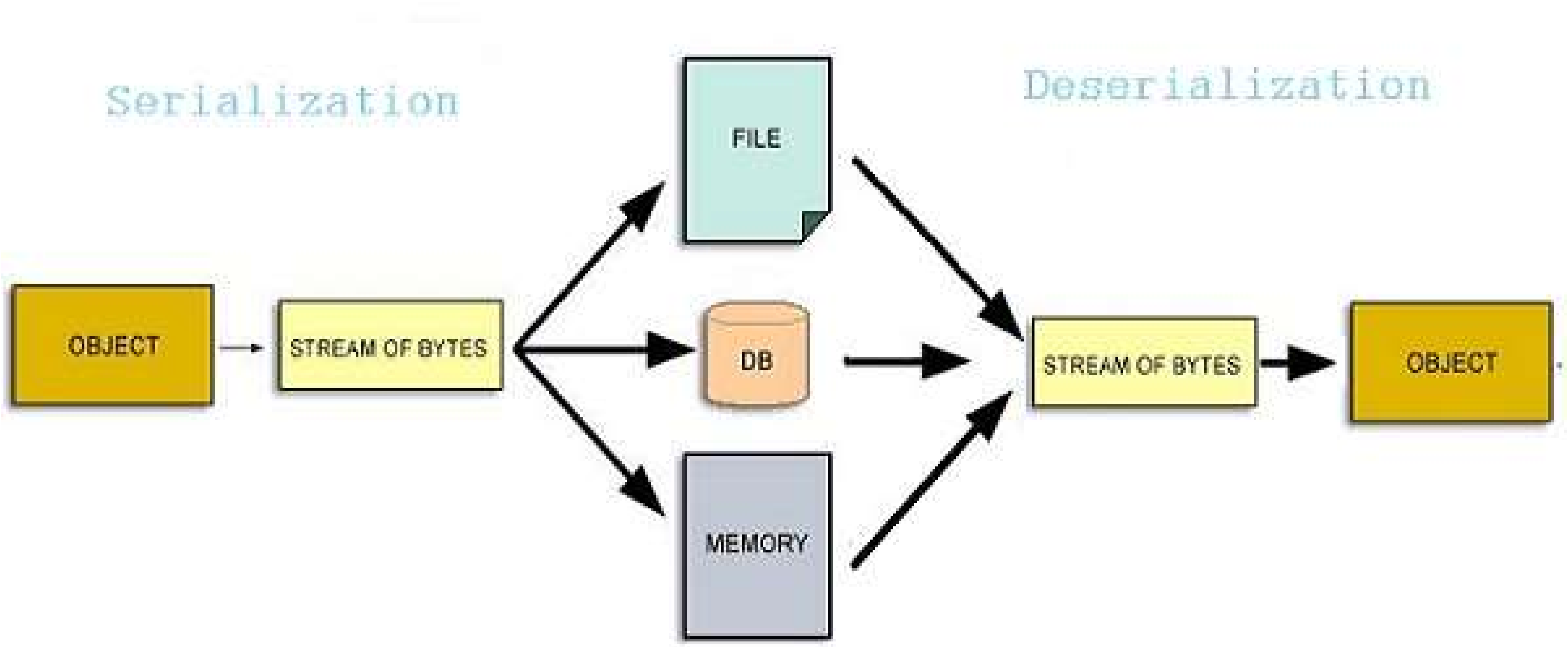
- HasExtension method enables you to determine if the path your application is processing has an extension.
- GetExtension method enables you to get the extension from a file name.
- GetTempFileName enables you to create a new temp file in your local Windows temporary directory in a single atomic operation folder.
  - This method then returns the absolute path to that file
- Path Class page at
  - http://go.microsoft.com/fwlink/?LinkID=267805

```
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
```

Lesson 2: Serializing and Deserializing Data

- What Is Serialization?
- Creating a Serializable Type
- Serializing Objects as Binary
- Serializing Objects as XML
- Serializing Objects as JSON
- Creating a Custom Serializer
- Demonstration: Serializing to XML

What Is Serialization?



https://www.studytonight.com/java/serialization-and-deserialization.php

What Is Serialization?

- **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.

- **Binary**
  - fast and lightweight, because the binary format does not require the processing and storage of unnecessary formatting constructs
  - enables you to preserve the fidelity and state of an object between different instances of your application
  - commonly used when persisting and transporting objects between applications running on the same platform.

```
1010101010101111101011
0101010110101111111010
10110110001
```

- **XML**
  - 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
  - less efficient and more processor intensive during the serializing, deserializing, and transporting processes
  - XML serialization is commonly used to serialize data that can be transported via the SOAP (simple object access protocol) protocol to and from web services

```
<SOAP-ENV:Envelope ...>
  <SOAP-ENV:Body>
    ...
      <ConfigName id="ref-3">
        FourthCoffee_Default
      </ConfigName>
    ...
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

- **JSON**
  - 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 is commonly used to transport data between Asynchronous JavaScript and XML (AJAX) calls because unlike XML, you are not limited to just communicating within the same domain.

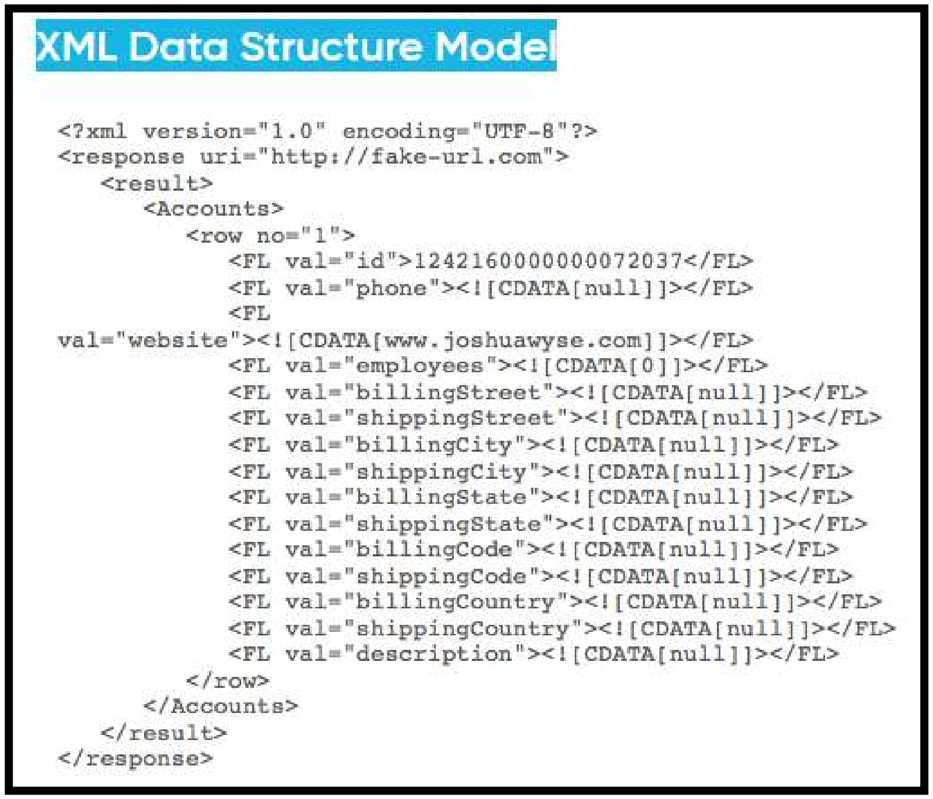
```
{
  "ConfigName":"FourthCoffee_Default",
  "DatabaseHostName":"database209.fourthcoffee.com"
}
```

- **Custom** ...
  - Create your own ...

Extra Read

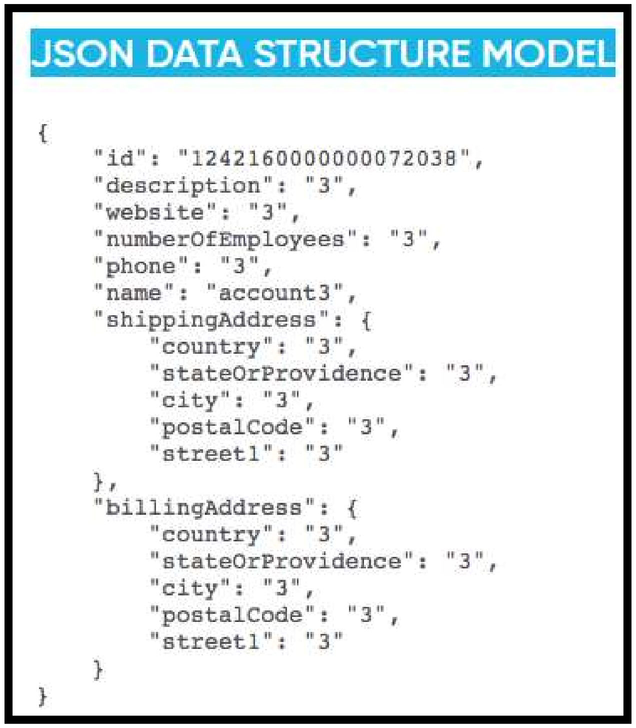
- You may want to read this blog post: …
  https://blog.cloud-elements.com/using-json-over-xml

- **Faster**: The **XML software parsing process** can take a long time. One reason for this problem is the DOM manipulation libraries that require more memory to handle large XML files. JSON uses less data overall, so you reduce the cost and increase the parsing speed.



XML Data Structure Model

```
<?xml version="1.0" encoding="UTF-8"?>
<response uri="http://fake-url.com">
   <result>
      <Accounts>
         <row no="1">
            <FL val="id">1242160000000072037</FL>
            <FL val="phone"><![CDATA[null]]></FL>
            <FL
val="website"><![CDATA[www.joshuawyse.com]]></FL>
            <FL val="employees"><![CDATA[0]]></FL>
            <FL val="billingStreet"><![CDATA[null]]></FL>
            <FL val="shippingStreet"><![CDATA[null]]></FL>
            <FL val="billingCity"><![CDATA[null]]></FL>
            <FL val="shippingCity"><![CDATA[null]]></FL>
            <FL val="billingState"><![CDATA[null]]></FL>
            <FL val="shippingState"><![CDATA[null]]></FL>
            <FL val="billingCode"><![CDATA[null]]></FL>
            <FL val="shippingCode"><![CDATA[null]]></FL>
            <FL val="billingCountry"><![CDATA[null]]></FL>
            <FL val="shippingCountry"><![CDATA[null]]></FL>
            <FL val="description"><![CDATA[null]]></FL>
         </row>
      </Accounts>
   </result>
</response>
```



JSON DATA STRUCTURE MODEL

```
{
    "id": "1242160000000072038",
    "description": "3",
    "website": "3",
    "numberOfEmployees": "3",
    "phone": "3",
    "name": "account3",
    "shippingAddress": {
        "country": "3",
        "stateOrProvidence": "3",
        "city": "3",
        "postalCode": "3",
        "street1": "3"
    },
    "billingAddress": {
        "country": "3",
        "stateOrProvidence": "3",
        "city": "3",
        "postalCode": "3",
        "street1": "3"
    }
}
```

Creating a Serializable Type

- To create your own serializable type, you need to do the following:

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

```
[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; }
...
```

Creating a Serializable Type – Example

```
[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

- To serialize your data to binary, you can use the **BinaryFormatter** class. The **BinaryFormatter** class implements the **IFormatter** interface.

- 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 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

- Serialize as XML

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

- Deserialize from XML

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

- 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.

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

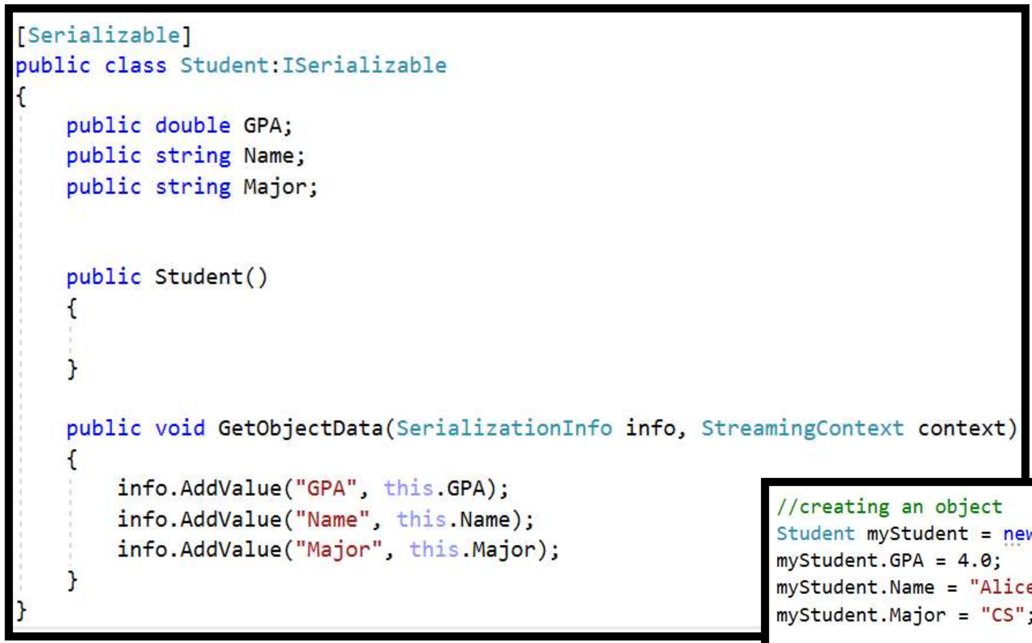
```
DataContractJsonSerializer jsonSerializer = new
   DataContractJsonSerializer(
      typeof(ServiceConfiguration));
FileStream buffer = File.OpenRead("C:\\fourthcoffee\\config.txt");
ServiceConfiguration config = jsonSerializer.ReadObject(buffer)
   as ServiceConfiguration;
buffer.Close();
```

- The .NET Framework also supports serializing objects as JSON by using the **DataContractJsonSerializer** class in the **System.Runtime.Serialization.Json** namespace.

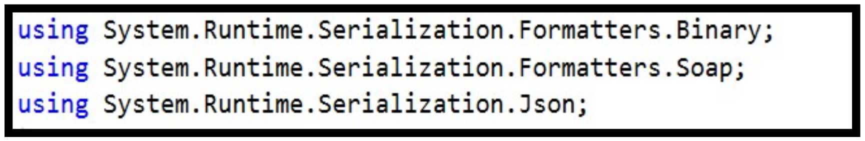
- The JSON serialization steps are different because the **DataContractJsonSerializer** class is derived from the abstract **XmlObjectSerializer** class, and it is not an implementation of the **IFormatter** interface

Example

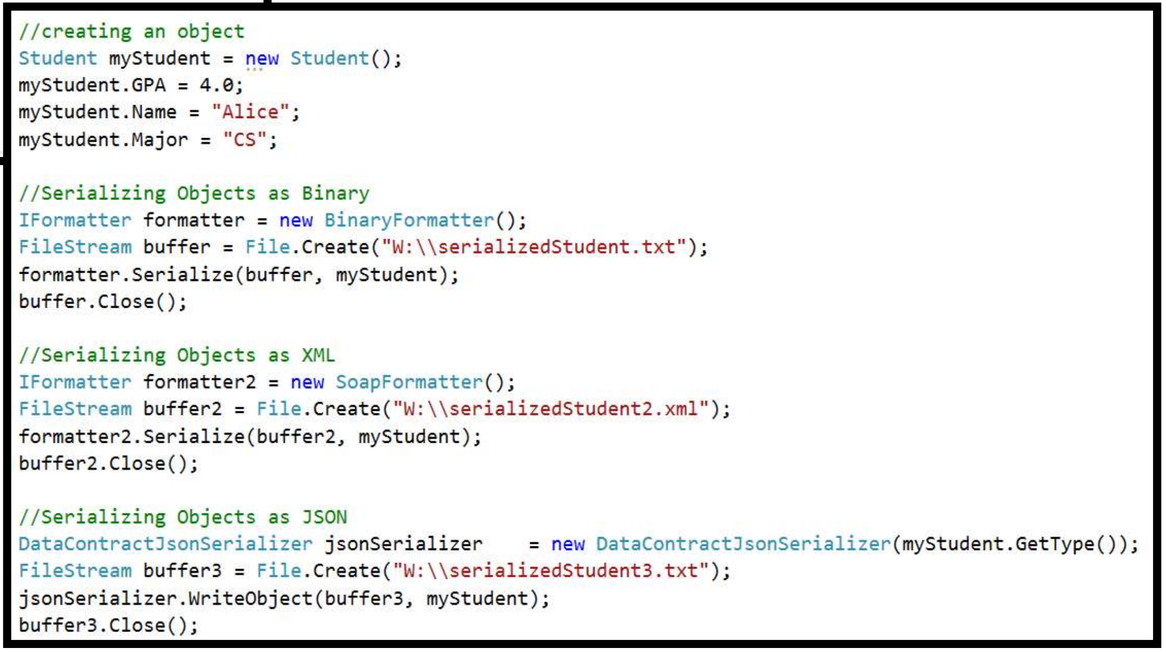
Create your own class and serialize it to a file using binary, XML, and JSON!!!



```
[Serializable]
public class Student:ISerializable
{
    public double GPA;
    public string Name;
    public string Major;


    public Student()
    {

    }

    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue("GPA", this.GPA);
        info.AddValue("Name", this.Name);
        info.AddValue("Major", this.Major);
    }
}
```



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



```
//creating an object
Student myStudent = new Student();
myStudent.GPA = 4.0;
myStudent.Name = "Alice";
myStudent.Major = "CS";

//Serializing Objects as Binary
IFormatter formatter = new BinaryFormatter();
FileStream buffer = File.Create("W:\\serializedStudent.txt");
formatter.Serialize(buffer, myStudent);
buffer.Close();

//Serializing Objects as XML
IFormatter formatter2 = new SoapFormatter();
FileStream buffer2 = File.Create("W:\\serializedStudent2.xml");
formatter2.Serialize(buffer2, myStudent);
buffer2.Close();

//Serializing Objects as JSON
DataContractJsonSerializer jsonSerializer    = new DataContractJsonSerializer(myStudent.GetType());
FileStream buffer3 = File.Create("W:\\serializedStudent3.txt");
jsonSerializer.WriteObject(buffer3, myStudent);
buffer3.Close();
```

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)
   {
      ...
   }
}
```

- To create your own formatter, you need to perform the following:

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.

Lesson 3: Performing I/O by Using Streams

- What are Streams?
- Types of Streams in the .NET Framework
- Reading and Writing Binary Data by Using Streams
- Reading and Writing Text Data by Using Streams
- Demonstration: Generating the Grades Report Lab

What are Streams?

- Reading and writing data in single **atomic operations** is acceptable when processing small files.
  - However, when you are working with large amounts of data, such operations are inefficient and can consume too much memory and processor time.

- A **stream** is a sequence of bytes (from a file, a network connection, or memory).
  - The .NET Framework provides the **Stream** base class in the **System.IO** namespace.
  - Streams enable you to read and write data in small, manageable chunks.
  - There are several streams classes that inherit from the **Stream** class, which provide streaming capabilities for different data types and storage mechanisms.

- The **System.IO namespace** contains a number of stream classes, including:
  - The abstract **Stream** base class
    - Internally, a Stream object maintains a pointer that refers to the current location in the data source
  - The **FileStream** class - uses a disk file as the data source
  - The **MemoryStream** class - uses a block of memory as the data source

- Typical stream operations include:
  - Reading chunks of data from a stream
  - Writing chunks of data to a stream
  - Querying the position of the stream

Types of Streams in the .NET Framework

- The .NET Framework provides many stream classes that enable you to read and write different types of data to and from different data sources.

- Classes that enable access to data sources include:
  - Raw sequence of bytes
  - Handles opening and closing (the file, the connection, etc.)

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:
  - textual data and primitive types

    to/from an underlying data source stream, such as a FileStream, MemoryStream, or NetworkStream object.

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.

Reading and Writing Binary Data by Using Streams

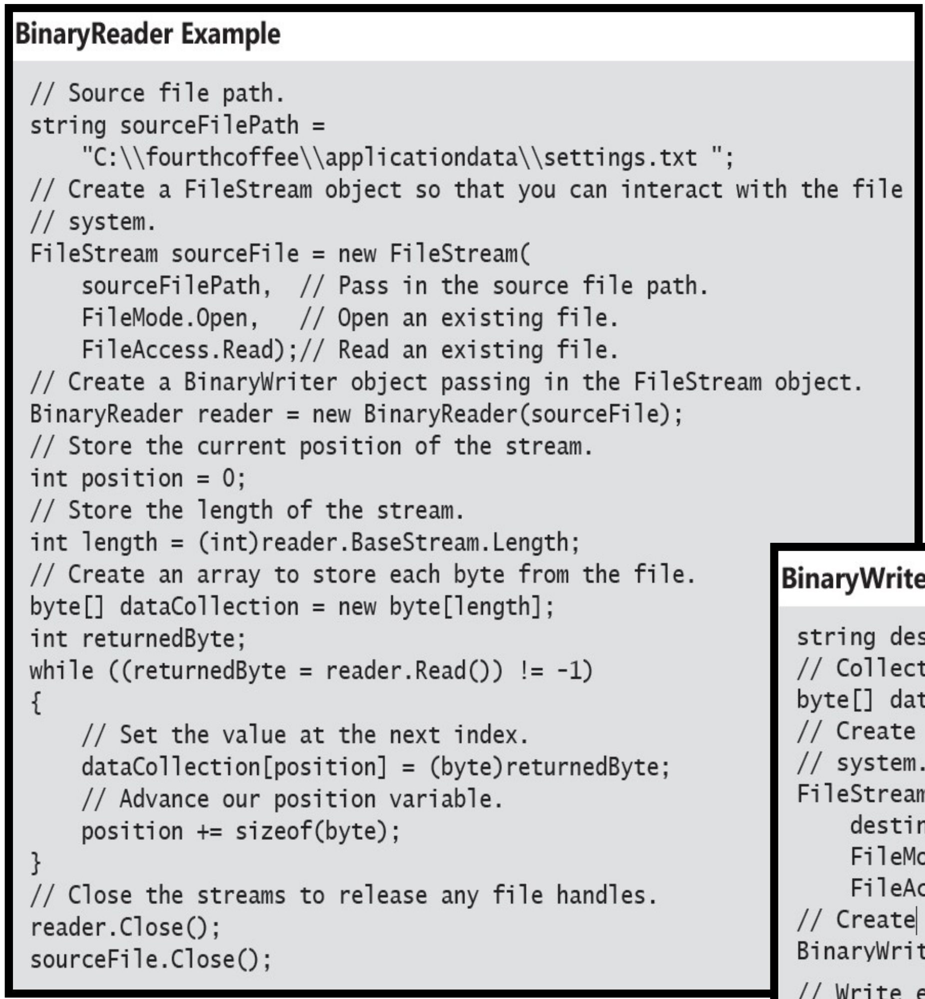
- Many applications store data in raw binary form because writing binary is fast, it takes up less space on disk, and because it is not human readable.

- 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);
```

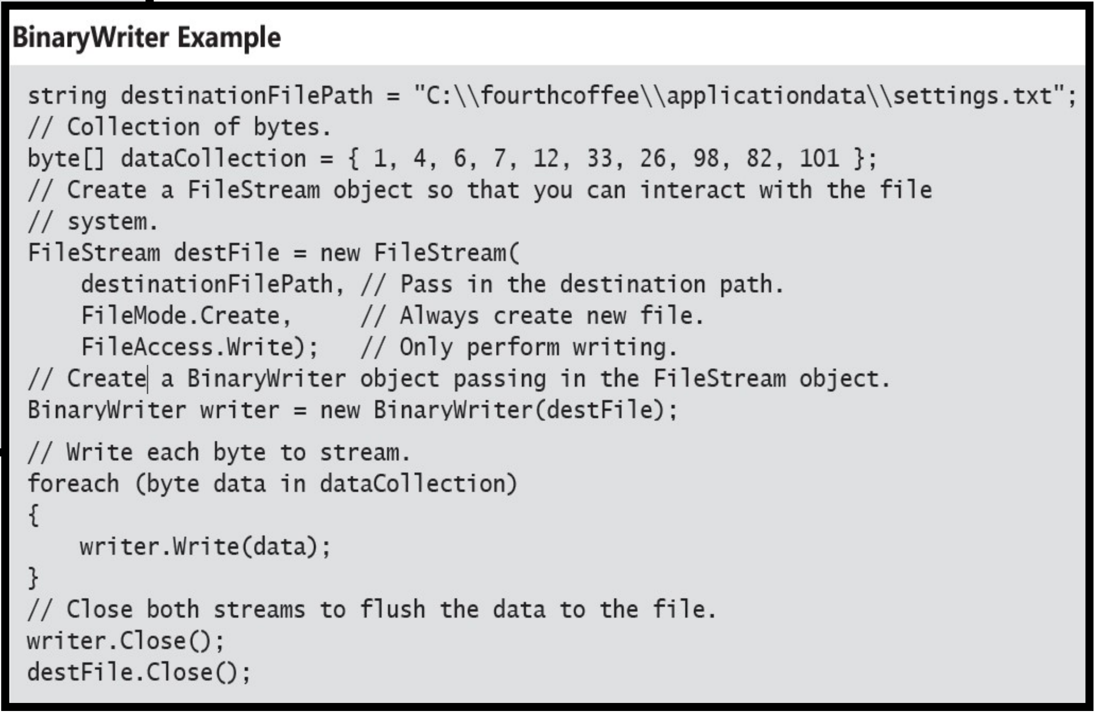
- After you have created a **BinaryReader** object
  - **BaseStream** property enables you to access the underlying stream that the BinaryReader object uses.
  - **Close** method enables you to close the BinaryReader object and the underlying stream.
  - **Read** method enables you to read the number of remaining bytes in the stream from a specific index.
  - **ReadByte** method enables you to read the next byte from the stream, and advance the stream to the next byte.
  - **ReadBytes** method enables you to read a specified number of bytes into a byte array
- Similarly, the **BinaryWriter** object ... write data to an underlying stream
  - **Flush** method enables you to explicitly flush any data in the current buffer to the underlying stream.
  - **Seek** method enables you to set your position in the current stream, thus writing to a specific byte.
  - **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 and Writing Binary Data by EXAMPLE



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();
```



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();
```

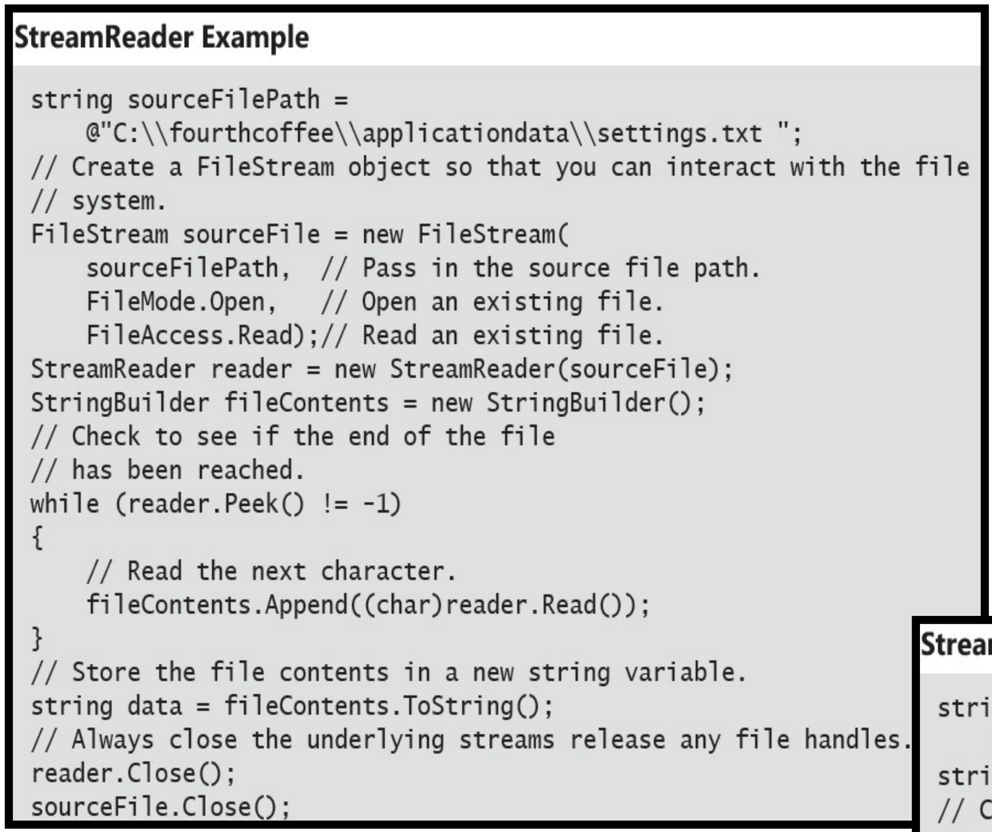
Reading and Writing Text Data by Using Streams

- Many applications store data as text, so that it is human readable and easier to process. The downside to this is that it takes up more space on disk.
- You can use the **StreamReader** and **StreamWriter** classes to stream plain text
  - When you initialize the StreamReader or StreamWriter classes, you must provide a stream object … data source.

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

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

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

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

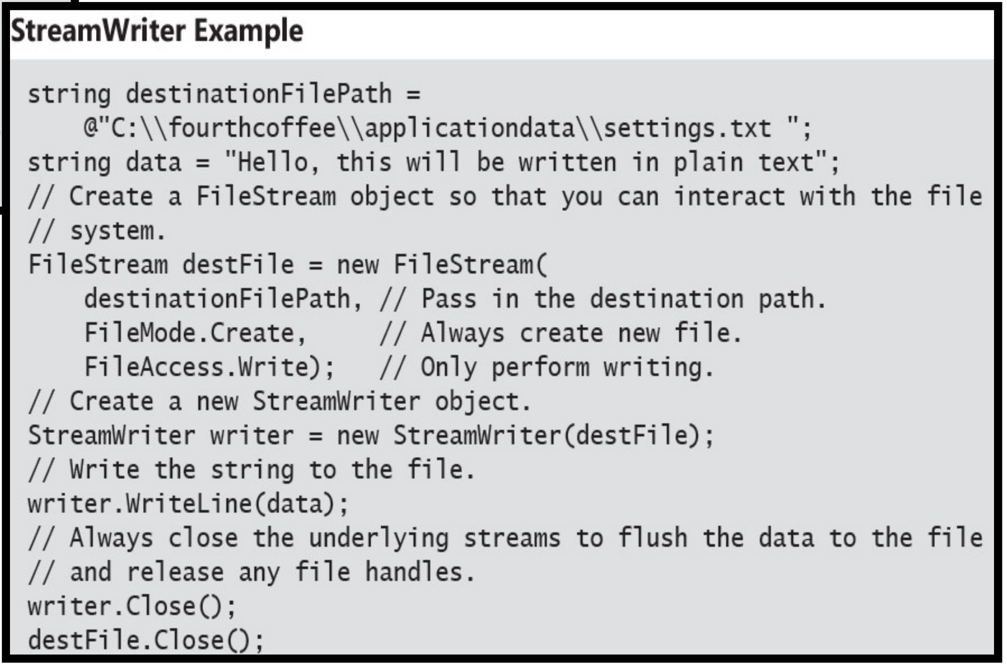
- After you have created a **StreamReader** object
  - **Close** method enables you to close the **StreamReader** object and the underlying stream.
  - **EndOfStream** property enables you to determine whether you have reached the end of the stream.
  - **Peek** method enables you to get the next available character in the stream, but does not consume it.
  - **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.
  - **ReadBlock** method enables you to read an entire block of characters from a specific index from the stream.
  - **ReadLine** method enables you to read an entire line of characters from the stream.
  - **ReadToEnd** method enables you to read all characters from the current position in the stream.
- **StreamWriter** object: AutoFlush, Close, Flush, NewLine, Write, WriteLine,…

Reading and Writing Text Data by Using Streams



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();
```



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();
```

Module Review and Takeaways

- **Question:** 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.

- **Question:** 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?

- **Question:** 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?
  - ( )Option 1: The MemoryStream, BinaryReader and BinaryWriter classes.
  - ( )Option 2: The FileStream, BinaryReader and BinaryWriter classes.
  - ( )Option 3: The BinaryReader and BinaryWriter classes.
  - ( )Option 4: The FileStream, StreamReader and StreamWriter classes.
  - ( )Option 5: The MemoryStream, StreamReader and StreamWriter classes.