CSc 346
Object Oriented Programming

Ken Gamradt

Spring 2024

Chapter 9 – Part 1

Working with Files, Streams, and Serialization

- This chapter is about reading and writing to files and streams, text encoding, and serialization
- Applications that do not interact with the filesystem are extraordinarily rare
- As a .NET developer, almost every application that you build will need to manage the filesystem and create, open, read, and write to and from files
- Most of those files will contain text, so it is important to understand how text is encoded
- And finally, after working with objects in memory, you will need to store them somewhere permanently for later reuse
- You do that using a technique called serialization

Managing the filesystem

- Your applications will often need to perform input and output operations with files and directories in different environments
- The **System** and **System.IO** namespaces contain classes for this purpose

Handling cross-platform environments and filesystems

- Let's explore how to handle cross-platform environments like the differences between Windows, macOS and Linux
- Paths are different for Windows, macOS, and Linux, so we will start by exploring how .NET handles this

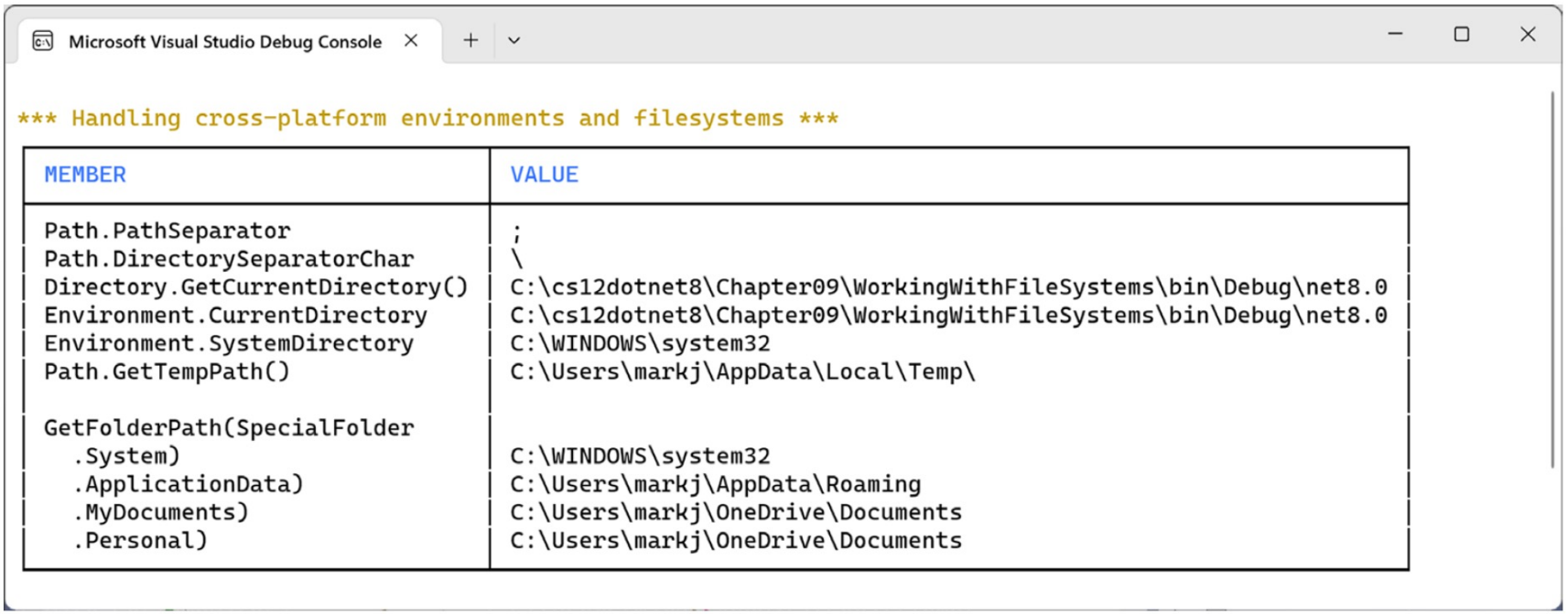
```
// Add to project file (.csproj)
<ItemGroup>
    <PackageReference Include="Spectre.Console" Version="0.47.0" />
</ItemGroup>

<ItemGroup>
    <Using Include="System.Console" Static="true" />
    <Using Include="System.IO.Directory" Static="true" />
    <Using Include="System.IO.Path" Static="true" />
    <Using Include="System.Environment" Static="true" />
</ItemGroup>
```

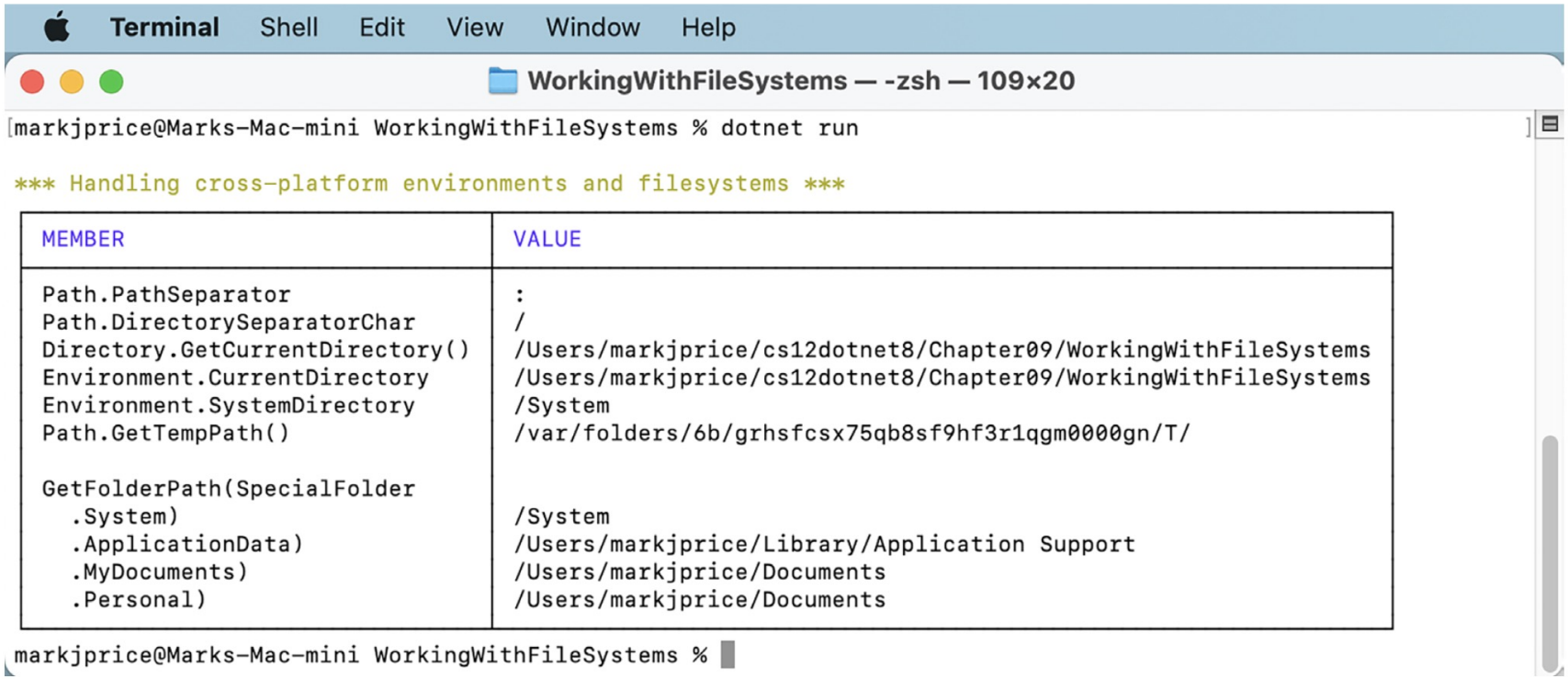
Handling cross-platform environments and filesystems

```
using Spectre.Console; // To use Table
#region Handling cross-platform environments and filesystems
SectionTitle("Handling cross-platform environments and filesystems");
Table table = new(); // Create a Spectre Console table
table.AddColumn("[blue]MEMBER[/]"); // Add two columns with markup for colors
table.AddColumn("[blue]VALUE[/]");
table.AddRow("Path.PathSeparator", PathSeparator.ToString()); // Add rows
table.AddRow("Path.DirectorySeparatorChar", DirectorySeparatorChar.ToString());
table.AddRow("Directory.GetCurrentDirectory()", GetCurrentDirectory());
table.AddRow("Environment.CurrentDirectory", CurrentDirectory);
table.AddRow("Environment.SystemDirectory", SystemDirectory);
table.AddRow("Path.GetTempPath()", GetTempPath());
table.AddRow("");
table.AddRow("GetFolderPath(SpecialFolder", "");
table.AddRow("  .System)", GetFolderPath(SpecialFolder.System));
table.AddRow("  .ApplicationData)", GetFolderPath(SpecialFolder.ApplicationData));
table.AddRow("  .MyDocuments)", GetFolderPath(SpecialFolder.MyDocuments));
table.AddRow("  .Personal)", GetFolderPath(SpecialFolder.Personal));
AnsiConsole.Write(table); // Render the table to the console
#endregion
```

Handling cross-platform environments and filesystems



Handling cross-platform environments and filesystems

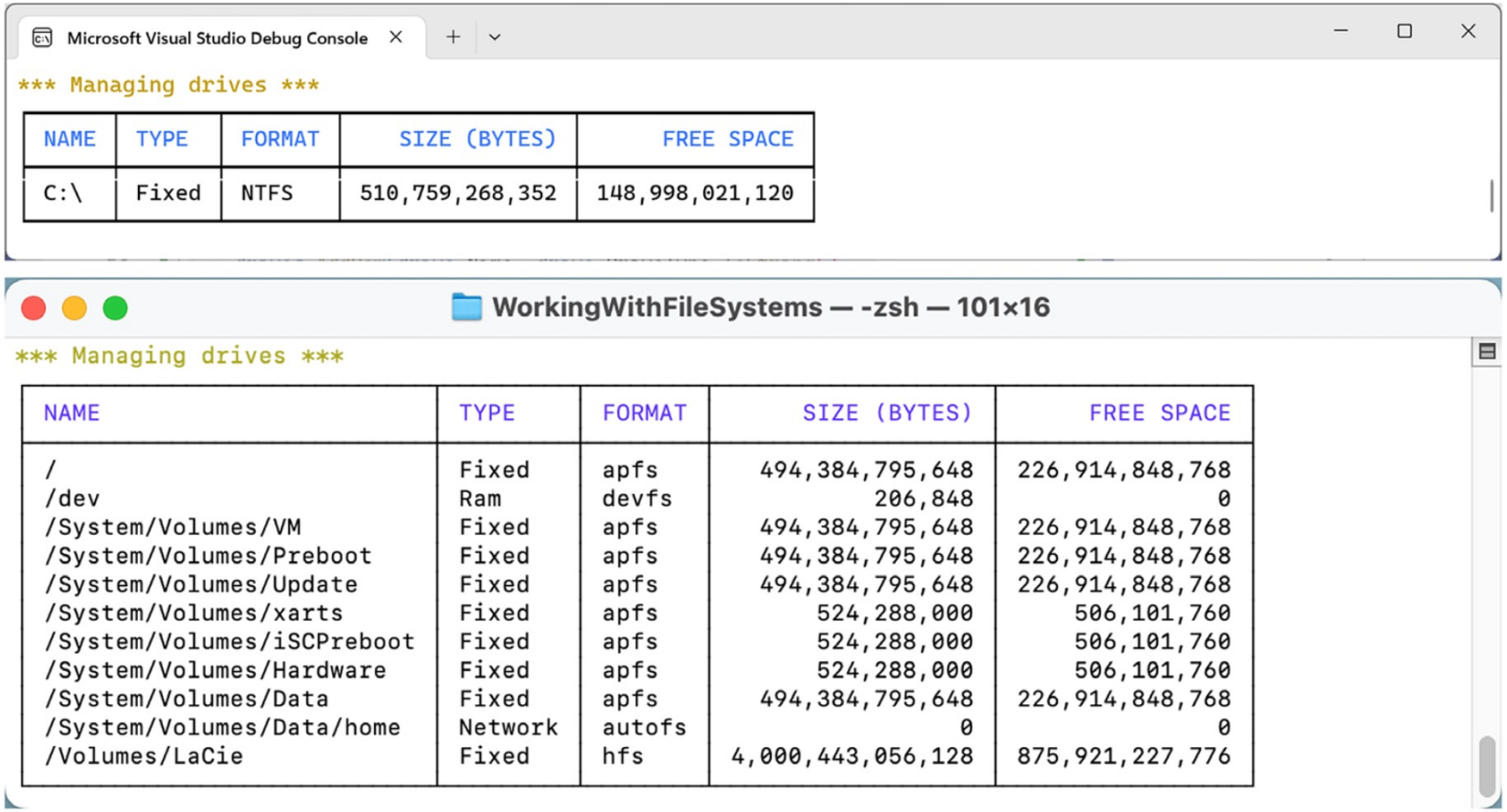


Managing drives

- To manage drives, use the **DriveInfo** type, which has a static method that returns information about all the drives connected to your computer. Each drive has a drive type

```
SectionTitle("Managing drives");
Table drives = new();
drives.AddColumn("[blue]NAME[/]");
drives.AddColumn("[blue]TYPE[/]");
drives.AddColumn("[blue]FORMAT[/]");
drives.AddColumn(new TableColumn("[blue]SIZE (BYTES)[/]").RightAligned());
drives.AddColumn(new TableColumn("[blue]FREE SPACE[/]").RightAligned());
foreach (DriveInfo drive in DriveInfo.GetDrives()) {
    if (drive.IsReady) {
        drives.AddRow(drive.Name, drive.DriveType.ToString(),
            drive.DriveFormat, drive.TotalSize.ToString("N0"),
            drive.AvailableFreeSpace.ToString("N0")
        );
    } else {
        drives.AddRow(drive.Name, drive.DriveType.ToString(),
            string.Empty, string.Empty, string.Empty);
    }
}
AnsiConsole.Write(drives);
```

Managing drives



Managing directories

- To manage directories, use the **Directory**, **Path**, and **Environment** static classes
- These types include many members for working with the filesystem
- When constructing custom paths, you must be careful to write your code so that it makes no assumptions about the platform
  - E.g., what to use for the directory separator character

Managing directories

```
SectionTitle("Managing directories");
string newFolder = Combine(GetFolderPath(SpecialFolder.Personal), "NewFolder");
WriteLine($"Working with: {newFolder}");
// We must explicitly say which Exists method to use
// because we statically imported both Path and Directory
WriteLine($"Does it exist? {Path.Exists(newFolder)}");
WriteLine("Creating it...");
CreateDirectory(newFolder);
// Let's use the Directory
// Exists method this time
WriteLine($"Does it exist? {Directory.Exists(newFolder)}");
Write("Confirm the directory exists, and then press any key.");
ReadKey(intercept: true);
WriteLine("Deleting it...");
Delete(newFolder, recursive: true);
WriteLine($"Does it exist? {Path.Exists(newFolder)}");
```

Managing directories

```
Working with: C:\Users\markj\OneDrive\Documents\NewFolder
Does it exist? False
Creating it...
Does it exist? True
Confirm the directory exists, and then press ENTER:
Deleting it...
Does it exist? False
```

Managing files

- When working with files, you could statically import the file type, just as we did for the directory type, but, for the next example, we will not, because it has some of the same methods as the directory type and they would conflict
- The file type has a short enough name not to matter in this case

Managing files

```
SectionTitle("Managing files");
// Define a directory path to output files starting in the user's folder
string dir = Combine(GetFolderPath(SpecialFolder.Personal), "OutputFiles");
CreateDirectory(dir);
// Define file paths
string textFile = Combine(dir, "Dummy.txt");
string backupFile = Combine(dir, "Dummy.bak");
WriteLine($"Working with: {textFile}");
WriteLine($"Does it exist? {File.Exists(textFile)}");
// Create a new text file and write a line to it
StreamWriter textWriter = File.CreateText(textFile); textWriter.WriteLine("Hello, C#!");
textWriter.Close(); // Close file and release resources
```

Managing files

```
WriteLine($"Does it exist? {File.Exists(textFile)}");
// Copy the file, and overwrite if it already exists
File.Copy(sourceFileName: textFile, destFileName: backupFile, overwrite: true);
WriteLine($"Does {backupFile} exist? {File.Exists(backupFile)}");
Write("Confirm the files exist, and then press any key.");
ReadKey(intercept: true);
// Delete the file
File.Delete(textFile);
WriteLine($"Does it exist? {File.Exists(textFile)}");
// Read from the text file backup
WriteLine($"Reading contents of {backupFile}:");
StreamReader textReader = File.OpenText(backupFile);
WriteLine(textReader.ReadToEnd());
textReader.Close();
```

Managing files

```
Working with: C:\Users\markj\OneDrive\Documents\OutputFiles\Dummy.txt
Does it exist? False
Does it exist? True
Does C:\Users\markj\OneDrive\Documents\OutputFiles\Dummy.bak exist? True
Confirm the files exist, and then press any key.
Does it exist? False
Reading contents of C:\Users\markj\OneDrive\Documents\OutputFiles\Dummy.
bak:
Hello, C#!
```

Managing paths

- Sometimes, you need to work with parts of a path; for example, you might want to extract just the folder name, the filename, or the extension. Sometimes, you need to generate temporary folders and filenames

```
SectionTitle("Managing paths");
WriteLine($"Folder Name: {GetDirectoryName(textFile)}");
WriteLine($"File Name: {GetFileName(textFile)}");
WriteLine("File Name without Extension: {0}", GetFileNameWithoutExtension(textFile));
WriteLine($"File Extension: {GetExtension(textFile)}");
WriteLine($"Random File Name: {GetRandomFileName()}");
WriteLine($"Temporary File Name: {GetTempFileName()}");
```

Managing paths

```
Folder Name: C:\Users\markj\OneDrive\Documents\OutputFiles
File Name: Dummy.txt
File Name without Extension: Dummy
File Extension: .txt
Random File Name: u45w1zki.co3
Temporary File Name:
C:\Users\markj\AppData\Local\Temp\tmphdmipz.tmp
```

Getting file information

- To get more information about a file or directory, e.g., its size or when it was last accessed, you can create an instance of the **FileInfo** or **DirectoryInfo** class
- FileInfo and DirectoryInfo both inherit from **FileSystemInfo**, so they both have members such as **LastAccessTime** and **Delete**, as well as extra members specific to themselves

Getting file information

Class	Members
FileSystemInfo	Fields: FullPath, OriginalPath Properties: Attributes, CreationTime, CreationTimeUtc, Exists, Extension, FullName, LastAccessTime, LastAccessTimeUtc, LastWriteTime, LastWriteTimeUtc, Name Methods: Delete, GetObjectData, Refresh
DirectoryInfo	Properties: Parent, Root Methods: Create, CreateSubdirectory, EnumerateDirectories, EnumerateFiles, EnumerateFileSystemInfos, GetAccessControl, GetDirectories, GetFiles, GetFileSystemInfos, MoveTo, SetAccessControl
FileInfo	Properties: Directory, DirectoryName, IsReadOnly, Length Methods: AppendText, CopyTo, Create, CreateText, Decrypt, Encrypt, GetAccessControl, MoveTo, Open, OpenRead, OpenText, OpenWrite, Replace, SetAccessControl

Controlling how you work with files

- When working with files, you often need to control how they are opened
- The **File.Open** method has overloads to specify additional options using enum values
  - **FileMode**: controls what you want to do with the file, like CreateNew, OpenOrCreate, or Truncate
  - **FileAccess**: controls what level of access you need, like ReadWrite
  - **FileShare**: controls locks on the file to allow other processes the specified level of access, like Read

Controlling how you work with files

- You might want to open a file and read from it (other processes to read also)

```
FileStream file =
    File.Open(pathToFile, FileMode.Open, FileAccess.Read, FileShare.Read);
```

- There is also an enum for attributes of a file as follows:
  - **FileAttributes**: check a FileSystemInfo-derived types' Attributes property for values like Archive and Encrypted

- You could check a file or directory's attributes, as shown in the following code:

```
FileInfo info = new(backupFile);
WriteLine("Is the backup file compressed? {0}",
          info.Attributes.HasFlag(FileAttributes.Compressed));
```

Reading and writing with streams

- A **stream** is a sequence of bytes that can be read from and written to
- Although files can be processed rather like arrays, with random access provided by knowing the position of a byte within the file, it can be useful to process files as a stream in which the bytes can be accessed in sequential order

- Streams can also be used to process terminal input and output and networking resources such as sockets and ports that do not provide random access and cannot seek (that is, move) to a position
- You can write code to process some arbitrary bytes without knowing or caring where it comes from
- Your code simply reads or writes to a stream, and another piece of code handles where the bytes are actually stored

Understanding abstract and concrete streams

- There is an **abstract class** named **Stream** that represents any type of stream
- There are many **concrete classes** that inherit from this base class, including
  - FileStream
  - MemoryStream
  - BufferedStream
  - GZipStream
  - SslStream

  so they all work the same way
- All streams implement IDisposable, so they have a Dispose method to release unmanaged resources

Understanding abstract and concrete streams

Member	Description
`CanRead`, `CanWrite`	These properties determine if you can read from and write to the stream.
`Length`, `Position`	These properties determine the total number of bytes and the current position within the stream. These properties may throw a `NotSupportedException` for some types of streams, for example, if `CanSeek` returns `false`.
`Close`, `Dispose`	This method closes the stream and releases its resources. You can call either method since the implementation of `Dispose` calls `Close`!
`Flush`	If the stream has a buffer, then this method writes the bytes in the buffer to the stream, and the buffer is cleared.
`CanSeek`	This property determines if the `Seek` method can be used.
`Seek`	This method moves the current position to the one specified in its parameter.
`Read`, `ReadAsync`	These methods read a specified number of bytes from the stream into a byte array and advance the position.
`ReadByte`	This method reads the next byte from the stream and advances the position.
`Write`, `WriteAsync`	These methods write the contents of a byte array into the stream.
`WriteByte`	This method writes a byte to the stream.

Understanding storage streams

Namespace	Class	Description
`System.IO`	`FileStream`	Bytes stored in the filesystem.
`System.IO`	`MemoryStream`	Bytes stored in memory in the current process.
`System.Net.Sockets`	`NetworkStream`	Bytes stored at a network location.

Understanding function streams

Namespace	Class	Description
`System.Security.Cryptography`	`CryptoStream`	This encrypts and decrypts the stream.
`System.IO.Compression`	`GZipStream`, `DeflateStream`	These compress and decompress the stream.
`System.Net.Security`	`AuthenticatedStream`	This sends credentials across the stream.

Understanding stream helpers

Namespace	Class	Description
`System.IO`	`StreamReader`	This reads from the underlying stream as plain text.
`System.IO`	`StreamWriter`	This writes to the underlying stream as plain text.
`System.IO`	`BinaryReader`	This reads from streams as .NET types. For example, the `ReadDecimal` method reads the next 16 bytes from the underlying stream as a `decimal` value and the `ReadInt32` method reads the next 4 bytes as an `int` value.
`System.IO`	`BinaryWriter`	This writes to streams as .NET types. For example, the `Write` method with a `decimal` parameter writes 16 bytes to the underlying stream and the `Write` method with an `int` parameter writes 4 bytes.
`System.Xml`	`XmlReader`	This reads from the underlying stream using XML format.
`System.Xml`	`XmlWriter`	This writes to the underlying stream using XML format.

Writing to text streams

- When you open a file to read or write to it, you use resources outside of .NET
- These are called **unmanaged resources** and must be disposed of when you are done working with them
- To deterministically control when they are disposed of, we can call the Dispose method
- When the Stream class was first designed, all cleanup code was expected to go in the Close method
- But later, the concept of IDisposable was added to .NET and Stream had to implement a Dispose method
- Later, the using statement was added to .NET, which can automatically call Dispose
- So today, you can call either Close or Dispose, and they actually do the same thing

Writing to text streams

```
0 references
public static class Viper {
    // Define an array of Viper pilot call signs
    0 references
    public static string[] Callsigns = new[] {
        "Husker", "Starbuck", "Apollo", "Boomer",
        "Bulldog", "Athena", "Helo", "Racetrack"
    };
}
```

Writing to text streams

```
SectionTitle("Writing to text streams");
// Define a file to write to
string textFile = Combine(CurrentDirectory, "streams.txt");
// Create a text file and return a helper writer
StreamWriter text = File.CreateText(textFile);
// Enumerate the strings, writing each one to the stream
// on a separate line
foreach (string item in Viper.Callsigns) {
    text.WriteLine(item);
}
text.Close(); // Release unmanaged file resources
OutputFileInfo(textFile);
```

Writing to text streams

```
**** File Info ****
File: streams.txt
Path: C:\cs12dotnet8\Chapter09\WorkingWithStreams\bin\Debug\net8.0
Size: 68 bytes.
/------------------
Husker
Starbuck
Apollo
Boomer
Bulldog
Athena
Helo
Racetrack

------------------/
```

Writing to XML streams

```
using System.Xml; // To use XmlWriter and so on
SectionTitle("Writing to XML streams"); // Define a file path to write to
string xmlFile = Combine(CurrentDirectory, "streams.xml");
// Declare variables for the filestream and XML writer
FileStream? xmlFileStream = null;
XmlWriter? xml = null;
try {
    xmlFileStream = File.Create(xmlFile);
    // Wrap the file stream in an XML writer helper and tell it
    // to automatically indent nested elements
    xml = XmlWriter.Create(xmlFileStream, new XmlWriterSettings { Indent = true });
    // Write the XML declaration
    xml.WriteStartDocument();
    // Write a root element
    xml.WriteStartElement("callsigns");
    // Enumerate the strings, writing each one to the stream
    foreach (string item in Viper.Callsigns) {
        xml.WriteElementString("callsign", item);
    }
    // Write the close root element.
    xml.WriteEndElement();
```

Writing to XML streams

```
} catch (Exception ex) {
    // If the path doesn't exist the exception will be caught
    WriteLine($"{ex.GetType()} says {ex.Message}");
} finally {
    if (xml is not null) {
        xml.Close();
        WriteLine("The XML writer's unmanaged resources have been disposed.");
    }
    if (xmlFileStream is not null) {
        xmlFileStream.Close();
        WriteLine("The file stream's unmanaged resources have been disposed.");
    }
}
OutputFileInfo(xmlFile);
```

Writing to XML streams

```
**** File Info ****
The XML writer's unmanaged resources have been disposed.
The file stream's unmanaged resources have been disposed.
File: streams.xml
Path: C:\cs12dotnet8\Chapter09\WorkingWithStreams\bin\Debug\net8.0
Size: 320 bytes.
/------------------
<?xml version="1.0" encoding="utf-8"?>
<callsigns>
  <callsign>Husker</callsign>
  <callsign>Starbuck</callsign>
  <callsign>Apollo</callsign>
  <callsign>Boomer</callsign>
  <callsign>Bulldog</callsign>
  <callsign>Athena</callsign>
  <callsign>Helo</callsign>
  <callsign>Racetrack</callsign>
</callsigns>
-------------------/
```

Simplifying disposal by using the using statement

- You can simplify the code that needs to check for a null object and then call its Dispose method by using the using statement
  - I would recommend using using rather than manually calling Dispose unless you need a greater level of control
- There are two uses for the **using** keyword:
  - importing a namespace
  - generating a finally statement that calls Dispose on an object that implements IDisposable
- The compiler changes a using statement block into a try-finally statement without a catch statement
- You can use nested try statements
  - So, if you do want to catch any exceptions, you can

Simplifying disposal by using the using statement

```
using (FileStream file2 = File.OpenWrite(Path.Combine(path, "file2.txt"))) {
    using (StreamWriter writer2 = new StreamWriter(file2)) {
        try {
            writer2.WriteLine("Welcome, .NET!");
        } catch(Exception ex) {
            WriteLine($"{ex.GetType()} says {ex.Message}");
        }
    } // Automatically calls Dispose if the object is not null
} // Automatically calls Dispose if the object is not null
```

Simplifying disposal by using the using statement

- You can even simplify the code further by not explicitly specifying the braces and indentation for the using statements

```
using FileStream file2 = File.OpenWrite(Path.Combine(path, "file2.txt"));
using StreamWriter writer2 = new(file2);
try {
    writer2.WriteLine("Welcome, .NET!");
} catch(Exception ex) {
    WriteLine($"{ex.GetType()} says {ex.Message}");
}
```