C# File I/O

Topics Covered

- File Pros & Cons
- The System.IO Namespace
- FileSystem Information Classes
- The File Class
  - Reading/Writing from Files
  - Move, Copy, Delete
- FileStream
  - Random Access to Files
- MemoryStream
  - Fixed-size and Expandable
- The Path Class

Using Files: Pros & Cons

We have numerous storage options these days – memory, files, clouds (Dropbox, Google, etc.), and databases for starts. Cloud storage requires coding to a particular API and is not considered in this list. Here are some of the pros/cons:

- **Persistence**: Unlike memory, data stored in files persists between sessions.
- **Speed**: File access is slower than memory access, but faster than database access.
- **Data Organization**: Databases facilitate organization of data, while files do not. As a result, random access from files can be much slower than database access.
- **Security**: Strong security layers can be defined for both files and databases.
- **Convenience**: File access is easy and convenient. Database access requires more preparation and a specialized language (SQL).

System.IO Namespace

.NET classes related to File I/O and general access to the file system are located in the System.IO namespace. A sample of the classes defined in this namespace are:

- DriveInfo, DirectoryInfo, FileInfo:These classes provide information about a drive, directory, or file.
- BinaryReader, BinaryWriter: These classes provide an interface for reading/writing built-in types as binary data from/to streams.
- FileStream, BufferedStream, MemoryStream: These are Stream subclasses. FileStream provides direct access to a file. BufferedStream provides a buffering layer to any underlying Stream. MemoryStream provides streamed access to memory.
- Directory: A static class providing directory-related methods.
- File: A static class providing File-related methods.
- FileSystemWatcher: Listens to the file system change notifications and raises events when a directory, or file in a directory, changes.
- Path:A static class providing methods to manipulate and prepare strings that represent paths to directories and files.

FileSystem Information Classes

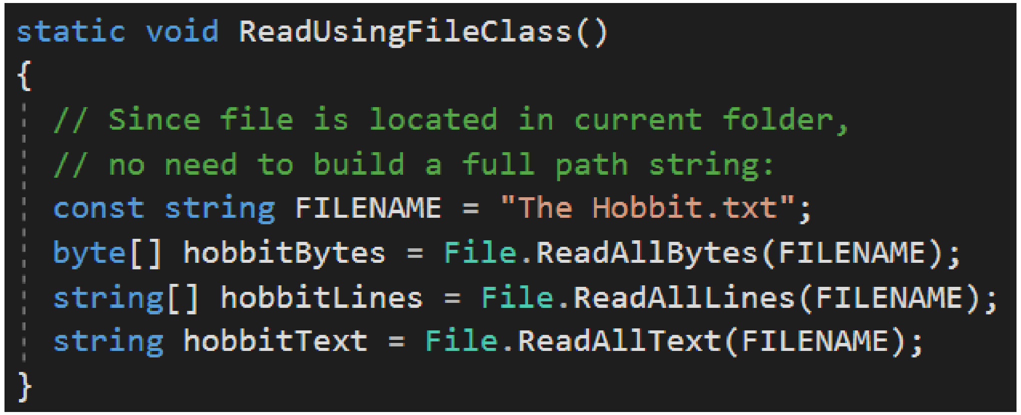
DriveInfo: Provides information about a drive. The static GetDrives method returns an array of DriveInfo representing every logical drive on a computer.

DirectoryInfo: Provides information about a directory, including whether it exists. Has methods for enumerating files, moving or creating subdirectories, and reading/writing access control entries (permissions).

FileInfo: Provides properties and methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of FileStream objects. Has methods for reading/writing access control entries (permissions).

Reading from Files with the File Class

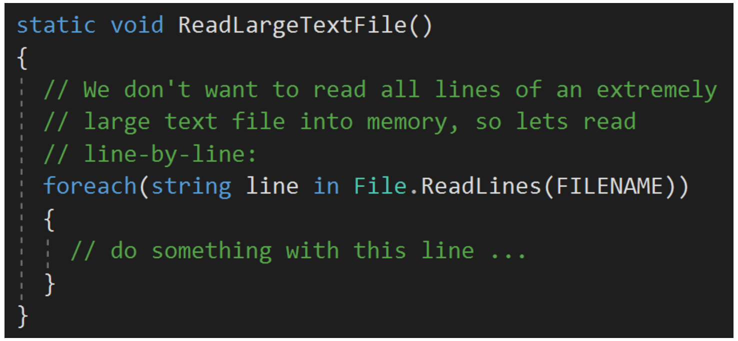
The File class provide static methods for quick access to files. It provides 3 methods for reading the contents of a file. **ReadAllBytes** returns the contents as binary data. **ReadAllText** returns the entire file as a string, and **ReadAllLines** returns the same content split based on newline characters:



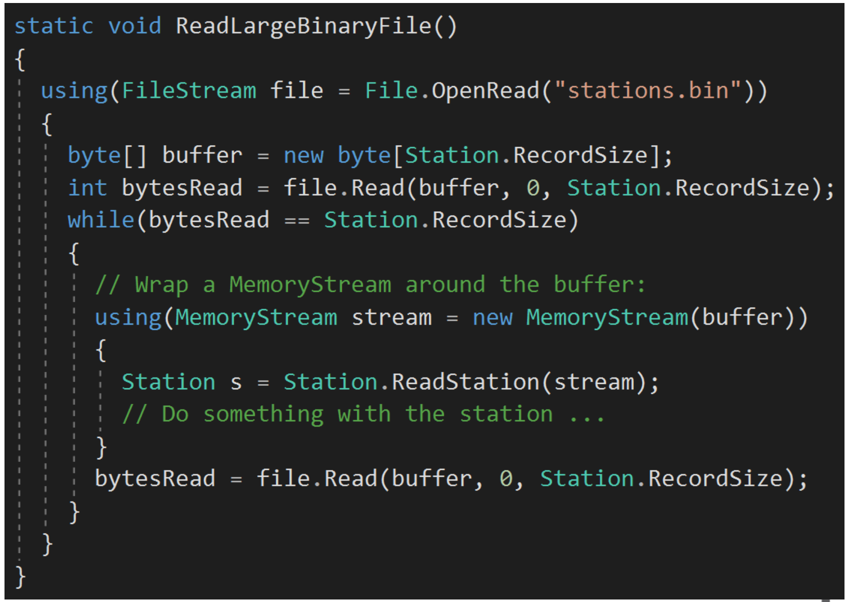
```
static void ReadUsingFileClass()
{
    // Since file is located in current folder,
    // no need to build a full path string:
    const string FILENAME = "The Hobbit.txt";
    byte[] hobbitBytes = File.ReadAllBytes(FILENAME);
    string[] hobbitLines = File.ReadAllLines(FILENAME);
    string hobbitText = File.ReadAllText(FILENAME);
}
```

Reading Large Files

Some files are quite large, so large that using the methods of the previous slide are inadvisable since each of those methods loads the entire contents of the file into memory. When working with large files, it is best to read and process text one line at a time, or use a buffer for binary data:



```
static void ReadLargeTextFile()
{
    // We don't want to read all lines of an extremely
    // large text file into memory, so lets read
    // line-by-line:
    foreach(string line in File.ReadLines(FILENAME))
    {
        // do something with this line ...
    }
}
```



```
static void ReadLargeBinaryFile()
{
    using(FileStream file = File.OpenRead("stations.bin"))
    {
        byte[] buffer = new byte[Station.RecordSize];
        int bytesRead = file.Read(buffer, 0, Station.RecordSize);
        while(bytesRead == Station.RecordSize)
        {
            // Wrap a MemoryStream around the buffer:
            using(MemoryStream stream = new MemoryStream(buffer))
            {
                Station s = Station.ReadStation(stream);
                // Do something with the station ...
            }
            bytesRead = file.Read(buffer, 0, Station.RecordSize);
        }
    }
}
```

Writing to Files with the File Class

The File class provides 3 **Write** methods analogous to the Read methods:

WriteAllBytes:	writes an array of bytes
WriteAllText:	writes a string
WriteAllLines:	writes an array of strings

There are also **Append** methods that append content to the end of a file:

AppendAllText:	appends a string
AppendAllLines:	appends an array of strings

Calling the Append methods in a fast loop can cause performance problems.

File also provides asynchronous methods for each of these operations.

File.OpenRead and File.OpenWrite return FileStream objects.

File.OpenText returns a StreamReader.

Other File Methods

Other methods provided by File are:

➢ Copy:	copy an existing file to a new location
➢ Exists:	test if a file exists
➢ Move:	move a file to a new location
➢ Open:	returns a FileStream to access the file contents
➢ OpenRead:	returns a read-only FileStream to access the file contents
➢ OpenWrite:	returns a write-only FileStream to access the file contents

This is not an exhaustive list. Please refer to the File documentation.

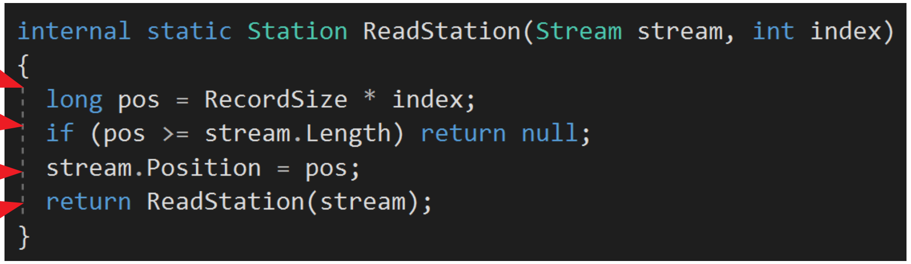
FileStream

The File class will probably satisfy 95% of your file-related tasks. FileStream has one notable advantage: its Position property is read/write, meaning that you can set the position from which to read or write information. This allows us to implement random access to fixed-length records.

An example is provided on the next slide.

FileStream Random Access

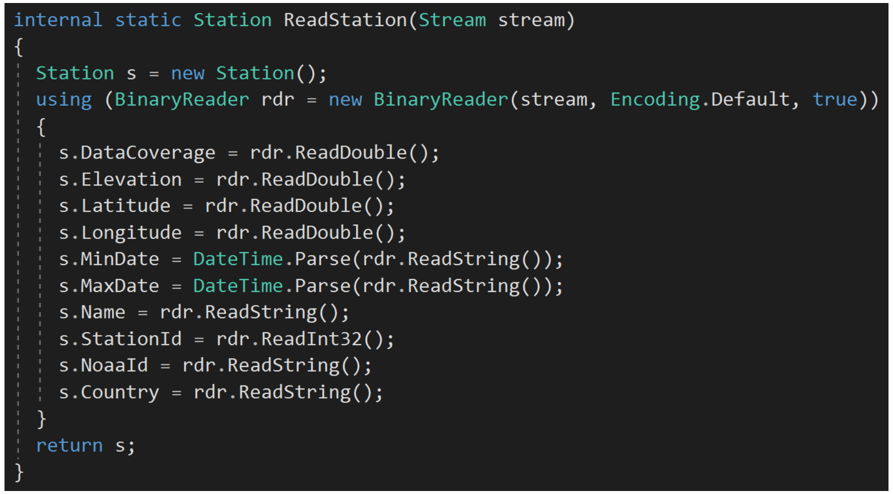
Calculate the read position.
Make sure it is within the file.
Set the read position.
Read the record.



```
internal static Station ReadStation(Stream stream, int index)
{
    long pos = RecordSize * index;
    if (pos >= stream.Length) return null;
    stream.Position = pos;
    return ReadStation(stream);
}
```

BinaryReader and BinaryWriter provide methods for reading/writing built-in types as binary data.

The last argument of this BinaryReader constructor ensures that the reader will leave the stream open.



```
internal static Station ReadStation(Stream stream)
{
    Station s = new Station();
    using (BinaryReader rdr = new BinaryReader(stream, Encoding.Default, true))
    {
        s.DataCoverage = rdr.ReadDouble();
        s.Elevation = rdr.ReadDouble();
        s.Latitude = rdr.ReadDouble();
        s.Longitude = rdr.ReadDouble();
        s.MinDate = DateTime.Parse(rdr.ReadString());
        s.MaxDate = DateTime.Parse(rdr.ReadString());
        s.Name = rdr.ReadString();
        s.StationId = rdr.ReadInt32();
        s.NoaaId = rdr.ReadString();
        s.Country = rdr.ReadString();
    }
    return s;
}
```

MemoryStream

MemoryStream provides a streaming interface to a chunk of memory. It can be constructed in several ways.

`MemoryStream()` This constructor creates a memory stream that will automatically allocate new new memory as bytes are written into it.

`MemoryStream(byte[])`, `MemoryStream(int)`, etc. These constructors create a fixed-length MemoryStream with optional arguments to make it read-only.

After writing information into a MemoryStream, you can acquire the byte array containing the data using MemoryStream.ToArray() or, with more caution, MemoryStream.GetBuffer().

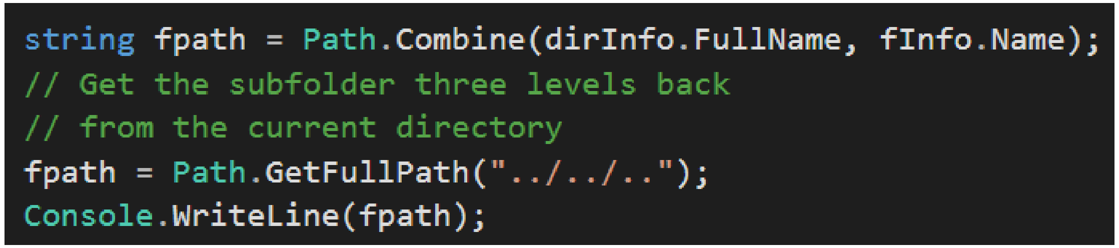
System.IO.Path

Path is a static class with methods for constructing and deconstructing path strings.

Path provides fields representing the characters (like ‘\’) used to build path strings.

Path.Combine combines folder names and file names, handling the logic of adding path separators where needed.

Path.GetDirectoryName, Path.GetFileName, Path.GetFileNameWithoutExtension do what the names imply.



```
string fpath = Path.Combine(dirInfo.FullName, fInfo.Name);
// Get the subfolder three levels back
// from the current directory
fpath = Path.GetFullPath("../../..");
Console.WriteLine(fpath);
```

Topics Covered

- File Pros & Cons
- The System.IO Namespace
- FileSystem Information Classes
- The File Class
  - Reading/Writing from Files
  - Move, Copy, Delete
- FileStream
  - Random Access to Files
- MemoryStream
  - Fixed-size and Expandable
- The Path Class

Questions

- What are alternatives to storing information in files? Describe some of the pros & cons of file-based storage.
- What is returned from these methods of the File class? ReadAllText; ReadAllBytes; ReadAllLines.
- If a file is extremely large, what are alternatives to using the above “All” methods?
- What are asynchronous methods?
- What can we do with a FileStream that we cannot do through the File class?
- What are the different ways we can create a MemoryStream?
- What are the differences between MemoryStream.ToArray() and MemoryStream.GetBuffer()?
- What is a path string?