Continue

last modified May 12, 2025 This tutorial explores file handling in C# and covers essential operations such as creating, reading, writing, deleting, and appending files. In C#, file management is handled using the System.IO and System.Text namespaces, providing efficient tools for working with files. The File class in System.IO offers static methods for fundamental file operations, including creation, copying, deletion, moving, and opening. It also facilitates the creation of FileStream objects for advanced file manipulation. For object-oriented file handling, .NET provides the FileInfo class. While FileInfo represents an individual file instance with detailed properties, File serves as a utility class for managing files across applications. The words.txt file This is a text file that we work with in some examples. words.txt sky blue cloud raisin tree falcon owl eagle rock water lake The File.create creates or overwrites a file in the specified path. It returns a FileStream, which provides read/write access to the file specified in path. Program.cs using System.Text; var path = "words.txt"; using FileStream fs = File.Create(path); byte[] data = Encoding.UTF8.GetBytes("falconribboncloud"); fs.Write(data, 0, data.Length); Console.WriteLine("data written to file"); The example creates a file and writes three words into it. This demonstrates how to use a FileStream to write raw bytes to a file, which is useful when you need precise control over file content or are working with binary data. using FileStream fs = File.Create(path); We create a file and retrieve a file stream to the file. This stream provides direct access to the file's bytes, allowing for efficient reading and writing operations. byte[] data = Encoding.UTF8.GetBytes("falconribboncloud"); We get the bytes of the three words we are going to write with Encoding.UTF8.GetBytes. This step is necessary because files store data as bytes, and converting strings to bytes ensures the correct encoding is used. fs.Write(data, 0, data.Length); The bytes are written to the file with Write. This method writes the specified byte array to the file, starting at the given offset and writing the specified number of bytes. C# File.CreateText The File.CreateText creates or opens a file for writing UTF-8 encoded text. If the file already exists, its contents are overwritten. It returns a StreamWriter that writes to the specified file using UTF-8 encoding. Program.cs var path = "words.txt"; using StreamWriter sw = File.CreateText(path); sw.WriteLine("falcon"); sw.WriteLine("sky"); sw.WriteLine("cloud"); Console.WriteLine("data written to file"); The example creates a new file with File.CreateText. It writes three words with WriteLine. This approach is convenient for writing text data, as the StreamWriter handles character encoding and line endings automatically. C# File.Copy The File.copy copies an existing file to a new file. It takes the source file and the destination file as parameters. Program.cs var sourcePath = "words.txt"; var destPath = "words_bck.txt"; File.Copy(sourcePath, destPath); Console.WriteLine("file copied"); The example copies a text file. This operation is useful for creating backups or duplicating files for further processing without altering the original. C# File.Move The File.Move moves a specified file to a new location. With this method, we can move a file to a new location or to rename a file. Program.cs var sourcePath = "words.txt"; var destPath = "data.txt"; File.Move(sourcePath, destPath); Console.WriteLine("file moved"); The example renames words.txt to data.txt. This demonstrates how to move a file to a new location or change its name within the same directory. C# File.Exists The File.Exists determines whether the specified file exists. Program.cs var path = "words2.txt"; if (File.Exists(path)) { Console.WriteLine("the file exists"); } else { Console.WriteLine("the file does not exist"); } In the code example we check if the words2.txt file exists. This is a common operation to prevent errors when attempting to access or modify files that may not be present. C# File.Delete The File.Delete deletes the specified file. Program.cs var path = "words.txt"; File.Delete(path); Console.WriteLine("file deleted"); The example deletes a text file. Deleting files is essential for managing disk space and removing outdated or unnecessary data from your application. C# File.GetCreationTime The File.GetCreationTime returns the creation date and time of the specified file or directory. Program.cs var path = "words.txt"; DateTime dt = File.GetCreationTime(path); Console.WriteLine($"Creation time: {dt}"); The example prints the creation time of the words.txt file. Knowing when a file was created can be useful for logging, auditing, or managing file lifecycles. $ dotnet run Creation time: 16. 1. 2024 11:31:35 C# File.GetLastWriteTime The File.GetLastWriteTime returns the date and time of the specified file or directory was last written to. Program.cs var path = "words.txt"; DateTime dt = File.GetLastWriteTime(path); Console.WriteLine($"Last write time: {dt}"); The example prints the last write time of the text file. This information helps track when a file was last modified, which is important for synchronization and backup tasks. C# File.Open The File.Open opens a FileStream on the specified path. The overloaded methods of File.Open allow to specify the file mode (Open, Create, CreateNew, Truncate, Append, or OpenOrCreate), the file access (Read, Write, ReadWrite). The file share value specifies the type of access other threads have to the file, such as Delete, None, Read, ReadWrite, Write, or Inheritable. Program.cs using System.Text; var path = "words.txt"; using FileStream fs = File.Open(path, FileMode.Open, FileAccess.Read); byte[] buf = new byte[1024]; int c; while ((c = fs.Read(buf, 0, buf.Length)) > 0) { Console.WriteLine(Encoding.UTF8.GetString(buf, 0, c)); } The example reads the contents of the words.txt file. This demonstrates how to use a FileStream to read raw bytes from a file and convert them into a string using a specific encoding. using FileStream fs = File.Open(path, FileMode.Open, FileAccess.Read); We open the file for reading. This ensures that the file is accessed in a way that prevents accidental modification while allowing data to be read efficiently. byte[] buf = new byte[1024]; int c; We create a buffer of bytes and an auxiliary variable. The buffer temporarily holds the data read from the file, and the variable tracks how many bytes were actually read. while ((c = fs.Read(buf, 0, buf.Length)) > 0) { Console.WriteLine(Encoding.UTF8.GetString(buf, 0, c)); } The Read method reads a block of bytes from the stream and writes the data to the given buffer. The Encoding.UTF8.GetString decodes a sequence of bytes into a string. This approach is efficient for reading large files in chunks. $ dotnet run sky blue cloud raisin tree falcon owl eagle rock water lake C# File.OpenRead The File.OpenRead opens an existing file for reading. It returns a read-only FileStream on the specified path. It is a convenience method to the File.Open. Program.cs using System.Text; var path = "words.txt"; using FileStream fs = File.OpenRead(path); byte[] buf = new byte[1024]; int c; while ((c = fs.Read(buf, 0, buf.Length)) > 0) { Console.WriteLine(Encoding.UTF8.GetString(buf, 0, c)); } The example reads a file with File.OpenRead. This method simplifies opening a file for reading by returning a read-only FileStream, making it easy to process file data. C# File.OpenText The File.OpenText opens an existing UTF-8 encoded text file for reading. It returns a StreamReader on the specified path. Program.cs var path = "words.txt"; using StreamReader sr = File.OpenText(path); string s = String.Empty; while ((s = sr.ReadLine()) != null) { Console.WriteLine(s); } The example opens a text file and reads its contents. Using a StreamReader allows you to read text data line by line, which is ideal for processing structured text files such as logs or CSVs. while ((s = sr.ReadLine()) != null) { Console.WriteLine(s); } We read the text file line by line. We do not have to decode the bytes into the string ourselves. The StreamReader handles the conversion from bytes to strings, making text processing straightforward. C# File.OpenWrite The File.OpenWrite opens an existing file or creates a new file for writing. It returns a FileStream object on the specified path with write access. Program.cs var path = "langs.txt"; using FileStream fs = File.OpenWrite(path); using StreamWriter sr = new StreamWriter(fs); sr.WriteLine("PHPDartJavaC#"); Console.WriteLine("data written"); The example opens a file in write mode and writes a line to the file. This demonstrates how to use a FileStream and StreamWriter together to write text data efficiently. using FileStream fs = File.OpenWrite(path); using StreamWriter sr = new StreamWriter(fs); We open a filestream to the specified path. The stream is passed to the StreamWriter, which is used to write characters in a particular encoding (UTF8 by default). This setup is useful for writing large amounts of text data to a file. sr.WriteLine("PHPDartJavaC#"); A line of text is written to the file with WriteLine. The StreamWriter automatically adds a newline character, making it easy to write multiple lines of text. C# File.ReadLines The File.ReadLines method reads lines from a text file lazily, returning an IEnumerable. Unlike File.ReadAllLines, it does not load all lines into memory at once, making it more efficient for large files. Program.cs var path = "words.txt"; var lines = File.ReadLines(path); foreach (var line in lines) { Console.WriteLine(line); } This example reads lines from a file one at a time using File.ReadLines, reducing memory usage compared to File.ReadAllLines. The returned enumerable is traversed with a foreach loop, processing each line as needed. This is especially useful for processing large files without loading them entirely into memory. C# File.ReadAllLines The File.ReadAllLines method reads all lines of a text file into a string[] array in a single operation. Since the entire file is loaded into memory at once, this approach is better suited for small to moderately-sized files. Program.cs var path = "words.txt"; string[] lines = File.ReadAllLines(path); foreach (var line in lines) { Console.WriteLine(line); } This example reads all lines from a file at once, storing them in a string[] array. The entire array is then iterated through using a foreach loop. This approach is suitable for small to moderately-sized files where memory usage is not a concern. If working with large files, consider using File.ReadLines to minimize memory consumption. This method reads lines lazily, making it more efficient for processing big files. C# File.ReadAllText The File.ReadAllText opens a text file, reads all the text in the file into a string, and then closes the file. Note that this method should not be used for very large files. Program.cs var path = "words.txt"; string readText = File.ReadAllText(path); Console.WriteLine(readText); The example reads the whole text into a string in one go. This is convenient for quickly loading the entire contents of a file, but should be avoided for very large files to prevent excessive memory usage. C# File.ReadAllBytes The File.ReadAllBytes opens a binary file, reads the contents of the file into a byte array, and then closes the file. Program.cs var path = "favicon.ico"; byte[] data = File.ReadAllBytes(path); int i = 0; foreach (byte c in data) { Console.Write("{0:X2} ", c); i++; if (i % 10 == 0) { Console.WriteLine(); } } The example reads a favicon.ico binary file. The data is printed to the console in hex format. This demonstrates how to process binary files and display their contents in a human-readable way. $ dotnet run 00 00 01 00 01 00 10 10 00 00 00 00 00 00 68 05 00 00 16 00 00 00 28 00 00 00 10 00 00 00 20 00 00 00 01 00 08 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 FF FF FF 00 4D 45 3D 00 00 00 00 00 00 00 ... C# File.WriteAllText The File.WriteAllText creates a new file, writes the contents to the file, and then closes the file. If the target file already exists, it is overwritten. Program.cs var path = "words.txt"; string data = "skycloudfalconowlcrane"; File.WriteAllText(path, data); Console.WriteLine("data written"); In the example, we write four words to a file with File.WriteAllText. This method is useful for quickly saving text data to a file, overwriting any existing content. C# File.WriteAllLines The File.WriteAllLines creates a new file, writes one or more strings to the file, and then closes the file. Program.cs using System.Text; var path = "words.txt"; string[] data = { "sky", "cloud", "falcon", "hawk" }; File.WriteAllLines(path, data, Encoding.UTF8); Console.WriteLine("data written"); In the example, we have an array of strings. We write these strings to a file in one go with File.WriteAllLines. This is an efficient way to save multiple lines of text to a file at once. C# File.WriteAllBytes The File.WriteAllBytes creates a new file, writes the specified byte array to the file, and then closes the file. Program.cs using System.Text; var path = "words.txt"; var text = "falconhawkforestcloudsky"; byte[] data = Encoding.UTF8.GetBytes(text); File.WriteAllBytes(path, data); Console.WriteLine("data written"); The example writes data to a file with File.WriteAllBytes. This method is ideal for saving binary data, such as images or serialized objects, directly to a file. var text = "falconhawkforestcloudsky"; byte[] data = Encoding.UTF8.GetBytes(text); First, we transform the text data into a array of bytes with Encoding.UTF8.GetBytes. This conversion ensures that the text is properly encoded for storage as binary data. File.WriteAllBytes(path, data); Then we write the array to the file with File.WriteAllBytes. This operation saves the entire byte array to the file in one step. C# File.AppendText The File.AppendText creates a StreamWriter that appends UTF-8 encoded text to an existing file, or to a new file if the specified file does not exist. Program.cs var path = "words.txt"; using StreamWriter sw = File.AppendText(path); sw.WriteLine("sky"); sw.WriteLine("lake"); The example appends two words to the words.txt file. Appending data is useful for adding new information to an existing file without overwriting its contents. C# File.AppendAllText The File.AppendAllText appends the specified string to the file. It creates the file if it does not exist. Program.cs var path = "words.txt"; var contents = "armourswordarrow"; File.AppendAllText(path, contents); Console.WriteLine("text appended to file"); The example appends three words to the specified text file. This method is convenient for adding text to a file, automatically creating the file if it does not exist. C# File.AppendAllLines The File.AppendAllLines appends lines to a file, and then closes the file. Program.cs var path = "words.txt"; List<string> data = ["brown", "blue", "khaki"]; File.AppendAllLines(path, data); Console.WriteLine("data written to file"); The example appends a list of strings to the file with File.AppendAllLines. This is helpful for adding multiple lines of text to a file in a single operation. C# File.Replace The File.Replace method replaces the contents of a specified file with the contents of another file, optionally creating a backup of the replaced file. Program.cs var sourceFile = "words.txt"; var destFile = "words2.txt"; var backupFile = "words_backup.txt"; File.Replace(sourceFile, destFile, backupFile); Console.WriteLine("file replaced"); The example replaces the contents of words2.txt with words.txt and creates a backup. This operation is useful for updating files while preserving the previous version for recovery if needed. C# File.SetAttributes and File.GetAttributes The File.SetAttributes and File.GetAttributes methods allow you to set and retrieve file attributes such as read-only or hidden. Program.cs var path = "words.txt"; File.SetAttributes(path, FileAttributes.ReadOnly); var attrs = File.GetAttributes(path); Console.WriteLine($"File attributes: {attrs}"); The example sets the file as read-only and then retrieves and prints its attributes. This demonstrates how you can programmatically change file properties and verify the changes, which is especially useful for automating file management tasks or enforcing certain file access policies in your C# programs. In this article we have worked with files in C#. We have utilized the File class of the System.IO. The provided examples cover a wide range of file operations, giving you practical knowledge to handle files efficiently and securely in your own C# projects. Source C# File class - language reference My name is Jan Bodnar, and I am a passionate programmer with extensive programming experience. I have been writing programming articles since 2007. To date, I have authored over 1,400 articles and 8 e-books. I possess more than ten years of experience in teaching programming. List all C# tutorials. The File class from the System.IO namespace, allows us to work with files: using System.IO; // include the System.IO namespace File.SomeFileMethod(); // use the file class with methods The File class has many useful methods for creating and getting information about files. For example: Method Description AppendText() Appends text at the end of an existing file Copy() Copies a file Create() Creates or overwrites a file Delete() Deletes a file Exists() Tests whether the file exists ReadAllText() Reads the contents of a file Replace() Replaces the contents of a file with the contents of another file WriteAllText() Creates a new file and writes the contents to it. If the file already exists, it will be overwritten. For a full list of File methods, go to Microsoft .Net File Class Reference. Write To a File and Read It In the following example, we use the WriteAllText() method to create a file named "filename.txt" and write some content to it. Then we use the ReadAllText() method to read the contents of the file: using System.IO; // include the System.IO namespace string writeText = "Hello World!"; // Create a text string File.WriteAllText("filename.txt", writeText); // Create a file and write the content of writeText to it string readText = File.ReadAllText("filename.txt"); // Read the contents of the file Console.WriteLine(readText); // Output the content The output will be: Hello World! Run example » File handling is a fundamental aspect of programming, and C# provides various methods to read from and write to files effectively. In this article, we will explore the best practices and efficient methods for file operations in C#, covering both reading and writing. For small files, File.ReadAllText and File.WriteAllText are simple and effective methods for reading and writing text content. using System;using System.IO;string filePath = "example.txt";string content = File.ReadAllText(filePath);Console.WriteLine("File Content:");Console.WriteLine(content); using System;using System.IO;string filePath = "example.txt";string content = "Hello, World!";File.WriteAllText(filePath, content);Console.WriteLine("File written successfully."); File.ReadAllText reads the entire content of a file into a single string. This method is suitable for smaller files, as it loads the entire file into memory. File.WriteAllText overwrites the file with the provided text, creating a new file if it does not exist. If you need to work with each line of a file separately, File.ReadAllLines and File.WriteAllLines are efficient and easy to use. using System;using System.IO;string filePath = "example.txt";string[] lines = File.ReadAllLines(filePath);Console.WriteLine("File Content by Line:");foreach (string line in lines){ Console.WriteLine(line);} using System;using System.IO;string filePath = "example.txt";string[] lines = { "Line 1", "Line 2", "Line 3" };File.WriteAllLines(filePath, lines);Console.WriteLine("File written with multiple lines successfully."); File.ReadAllLines reads each line of the file into a string array, making it easy to iterate over lines. File.WriteAllLines writes an array of strings to a file, with each element representing a line. For large files, StreamReader and StreamWriter allow efficient reading and writing by handling data in streams rather than loading the entire file into memory. using System;using System.IO;string filePath = "largeFile.txt";using (StreamReader reader = new StreamReader(filePath)){ string line; while ((line = reader.ReadLine()) != null) { Console.WriteLine(line); }} using System;using System.IO;string filePath = "largeFile.txt";using (StreamWriter writer = new StreamWriter(filePath)){ writer.WriteLine("This is the first line."); writer.WriteLine("This is the second line.");}Console.WriteLine("Large file written successfully."); StreamReader reads each line individually, making it memory-efficient for large files. StreamWriter writes data line by line to the file, which is suitable for handling large volumes of data. Using using statements with StreamReader and StreamWriter ensures that file resources are properly disposed of when the operation is complete. When you need byte-level control over file operations or are working with binary data, FileStream is the most efficient approach. It provides low-level access to files. using System;using System.IO;string filePath = "binaryFile.dat";using (FileStream fs = new FileStream(filePath, FileMode.Open, FileAccess.Read)){ byte[] buffer = new byte[fs.Length]; fs.Read(buffer, 0, buffer.Length); Console.WriteLine("Binary file content:"); foreach (byte b in buffer) { Console.Write(b + " "); }} using System;using System.IO;string filePath = "binaryFile.dat";byte[] data = { 0x01, 0x02, 0x03, 0x04 };using (FileStream fs = new FileStream(filePath, FileMode.Create, FileAccess.Write)){ fs.Write(data, 0, data.Length);}Console.WriteLine("Binary data written successfully."); FileStream reads and writes data as bytes, which is ideal for binary files or when you need precise control over file structure. FileMode and FileAccess parameters allow for flexibility in how the file is accessed (e.g., FileMode.Create creates a new file). BufferedStream can be layered over FileStream to improve performance in cases where you perform many small read or write operations. using System;using System.IO;string filePath = "largeFile.dat";using (FileStream fs = new FileStream(filePath, FileMode.OpenOrCreate))using (BufferedStream bs = new BufferedStream(fs)){ byte[] buffer = new byte[1024]; int bytesRead; while ((bytesRead = bs.Read(buffer, 0, buffer.Length)) > 0) { Console.WriteLine($"Read {bytesRead} bytes from file."); }} BufferedStream can improve the speed of file operations by reducing the number of actual read or write operations to the disk. This can be beneficial for files with frequent small reads or writes. MethodBest ForMemory UsageCreates StreamFile.ReadAllText / File.WriteAllTextSmall files with text contentModerateNoFile.ReadAllLines / File.WriteAllLinesSmall files line by lineModerateNoStreamReader / StreamWriterLarge files with text contentLowYesFileStreamBinary data or byte-level controlLowYesBufferedStreamHigh-frequency small reads/writesLowYes In C#, there are multiple ways to efficiently read and write files depending on your requirements. For small files, File.ReadAllText or File.WriteAllText are convenient and fast, while StreamReader and StreamWriter are better suited for larger files. For binary data or byte-level control, FileStream and BufferedStream provide optimized handling. Choose the method that best fits your use case to ensure efficient file handling in C#. Download "C# Essentials: A Developer's Cheat Sheet" for key syntax, tips, and quick references. Perfect for developers of all levels! Download your free copy now! We offer expert support and development services for projects of any size. Contact us for a free consultation and see how we can help you succeed. CONTACT US NOW There are a few ways to create a file and write to it using the .NET File API (in System.IO). The simplest way is to use high-level methods like File.WriteAllText() and File.WriteAllLines(), specifying the file path and string(s) to write to the file. Here's an example of using these (and their async equivalents): using System.IO; File.WriteAllText(@"C:\temp\a.txt", "Hello World"); var lines = new List() { "Hello", "World" }; File.WriteAllLines(@"C:\temp\b.txt", lines); await File.WriteAllTextAsync(@"C:\temp\a_async.txt", "Hello World Async"); await File.WriteAllLinesAsync(@"C:\temp\b_async.txt", lines); Code language: C# (cs) These high-level methods abstract away the details. They create the file, open it for writing, write the content, and then close the file. If the file already exists, they overwrite it. The next best option is to use File.CreateText(). This returns a StreamWriter, which you can use to write to the file. Here's an example: using System.IO; using (StreamWriter writer = File.CreateText(@"C:\temp\b.txt")) { writer.WriteLine("Hello World"); writer.WriteLine(7); writer.WriteLine(true); } Code language: C# (cs) This creates the text file, opens it for writing, writes each line, and then closes the file (when it falls out of the using block). If the file already exists, this overwrites it. Here's what the file content looks like: Hello World 7 True Code language: plaintext (plaintext) I'll show examples of less common scenarios (like writing to a binary file), and also discuss some problems to avoid. If you try to create a file in a non-existing directory, you'll get an exception like this: System.IO.DirectoryNotFoundException: Could not find a part of the path 'C:\data\en\hello.txt'. You can use Directory.CreateDirectory() to create the missing directories. Pass in the directory path (without the file name), and it'll create all missing directories in the path. Here's an example (C:\data\ and C:\data\en\ are missing): using System.IO; var filePath = @"C:\data\en\hello.txt"; var directory = Path.GetDirectoryName(filePath); Directory.CreateDirectory(directory); File.WriteAllText(filePath, "Hello new directory"); Code language: C# (cs) This first creates the C:\data\ and C:\data\en\ directories, and then creates the file in C:\data\en\ and writes to it. Note: If the directory already exists, Directory.CreateDirectory() does nothing. You can specify a relative or absolute file path when creating a file. If you specify just a file name, it'll create the file in the current directory (whatever Environment.CurrentDirectory is set to). Here's an example: using System.IO; Console.WriteLine("Current directory:"); Console.WriteLine(Environment.CurrentDirectory); File.WriteAllText("new.txt", "Created in cur dir"); var filePath = Path.Combine(Environment.CurrentDirectory, "new.txt"); Console.WriteLine(); Console.WriteLine($"Does the file exist? {File.Exists(filePath)}"); Code language: C# (cs) This outputs: Current directory: D:\Project\bin\Debuget6.0 Does the file exist? TrueCode language: plaintext (plaintext) Most of the time you'll be dealing with text files, but sometimes you may need to create and write to binary files. The two simplest ways to do this are to use File.WriteAllBytes() or use File.Create() and BinaryWriter. These are equivalent to the text file methods I showed up above. Here's an example of writing a byte array to a binary file with File.WriteAllBytes(): using System.IO; byte[] data = new byte[] { 0b0110_1000, 0b0110_0101, 0b0110_1100, 0b0110_1100, 0b0110_1111 }; File.WriteAllBytes(@"C:\temp\data.bin", data); Code language: C# (cs) This creates a binary file with the ASCII bytes for "hello" in it. The other option is to use File.Create() to create the file. This returns a FileStream. You can then use BinaryWriter to write data to the FileStream. Here's an example: using System.IO; using (var writer = new BinaryWriter(File.Create(@"C:\temp\hello.bin"))) { writer.Write("Hello World"); writer.Write(true); writer.Write(10); writer.Write(7.5m); } Code language: C# (cs) This creates a binary file and writes bytes for whatever objects you pass it. Note: If you open it up in Notepad++, you can see the unprintable characters, like [NUL]. Opening files in C# involves types from the System.IO namespace. Methods like File.ReadAllText can easily read in a file. Often loops are not even needed.Writing to files can be done with methods like File.WriteAllText. More advanced methods, like StreamReader and StreamWriter, are often better choices.StreamReaderFor text files, StreamReader and StreamWriter are often the most useful types. StreamReader introduces some complexity in our use of the language—we see the "using" statement.Step 1 Here we open the file for reading. We use StreamReader in a "using" block—this allows automatic cleanup of resources.Step 2 We call ReadLine. This is a method on StreamReader. It returns null if no further data is available in the file.Step 3 Here we have the line variable. This contains a line of the file (with no newlines included).using System; using System.IO; class Program { static void Main() { // Step 1: open file for reading. using (StreamReader reader = new StreamReader(@"C:\programs\file.txt")) { // Step 2: call ReadLine until null. string line; while ((line = reader.ReadLine()) != null) { // Step 3: do something with the line. Console.WriteLine($"LINE: {line}"); } } } }LINE: Hello my friend LINE: Welcome to the Internet LINE: Third line in fileStreamWriterThis class writes strings or append strings to a text file. We can write numbers or the textual representation of anything. It also uses a "using" block.using System.IO; class Program { static void Main() { // Create or open file and write line to it. // ... If file exists, it contents are erased before writing. using (var writer = new StreamWriter(@"C:\programs\example.txt")) { writer.WriteLine("HELLO"); } } }ReadAllTextThis program uses this method to load in the file "file.txt" on the C: volume. Then it prints the contents of the file. The data is now stored in a string object.Tip ReadAllText is the easiest way to put a file into a string. It is part of the System.IO namespace.using System; using System.IO; class Program { static void Main() { string file = File.ReadAllText("C:\\file.txt"); Console.WriteLine(file); } }ReadAllLinesHere we read all the lines from a file and place them in an array. The code reads lines from "file.txt" and uses a foreach-loop on them. This is efficient code.using System.IO; class Program { static void Main() { // Read in every line in specified file. // ... This will store all lines in an array in memory. string[] lines = File.ReadAllLines("file.txt"); foreach (string line in lines) { // Do something with the line. if (line.Length > 80) { // Important code. } } } }Count linesWe count the number of lines in a file with few lines of code. The example here is a bit slow. But it works. It references the Length property.using System.IO; class Program { static void Main() { // Another method of counting lines in a file. // ... This is not the most efficient way. // ... It counts empty lines. int lineCount = File.ReadAllLines("file.txt").Length; } }QueryDoes a line containing a specific string exist in the file? Maybe we want to see if a name or location exists in a line in the file. We use LINQ to find any matching line.using System.IO; using System.Linq; class Program { static void Main() { // See if line exists in a file. // ... Use a query expression to count matching lines. // ... If one matches, the bool is set to true. bool exists = (from line in File.ReadAllLines("file.txt") where line == "Some line match" select line).Count() > 0; } }ReadLinesThis method does not immediately read in every line. It instead reads lines only as they are needed. We use it in a foreach-loop.using System; using System.IO; class Program { static void Main() { // Read lines in file 1-by-1. foreach (string line in File.ReadLines(@"C:\programs\file.txt")) { Console.WriteLine("LINE: {0}", line); } } }LINE: Hello my friend LINE: Welcome to the Internet ...WriteAllLinesWe can write an array to a file. When we are done within-memory processing, we often need to write the data to disk.using System.IO; class Program { static void Main() { // Write a string array to a file. string[] stringArray = new string[] { "cat", "dog", "arrow" }; File.WriteAllLines("file.txt", stringArray); } }cat dog arrowWriteAllTextA simple method, File.WriteAllText receives two arguments. It receives the path of the output file, and the exact string contents of the text file.Note The file is created if it does not exist, or replaced with a new file if it does exist (no appends ever occur).using System.IO; class Program { static void Main() { File.WriteAllText("C:\\perls.txt", "Dot Net Perls"); } }AppendAllTextWe could read in a file, append to that in memory, and then write it out completely again. That is slow—it's more efficient to use an append.Argument 1 The first argument to File.AppendAllText is the name of the file we wish to append text to.Argument 2 The second argument is the string we wish to append to the file—we must add a newline at the end if we want to write a line.Detail Inspected in IL Disassembler: AppendAllText internally calls StreamWriter, with the second parameter "append" set to true.Note If the file already exists when the program starts, the file will be appended to. Otherwise, a new file is created.using System.IO; class Program { static void Main() { // Use AppendAllText to write one line to the text file. File.AppendAllText("C:\\perls.txt", "first part"); // The file now has a newline at the end, so write another line. File.AppendAllText("C:\\perls.txt", "second part"); // Write a third line. string third = "third part"; File.AppendAllText("C:\\perls.txt", third); } }first part second part third partfirst part second part third part first part second part third partReadAllBytesWe use File.ReadAllBytes to read an image into memory. Here we read the bytes from a WEBP file into a byte array, and print its length and the first byte.using System; using System.IO; class Program { static void Main() { // Read in a binary file. byte[] webpFile = File.ReadAllBytes(@"C:\programs\test.webp"); Console.WriteLine("Length: {0}", webpFile.Length); Console.WriteLine("First byte: {0}", webpFile[0]); } }Length: 822 First byte: 82Benchmark, ReadLineSuppose we have a text file with many lines—possibly 1 million lines. We could use a method like ReadAllLines, or StreamReader on the file.Version 1 In this version of the code, we read in the file with StreamReader line-by-line.Version 2 Here we read in the file with File.ReadAllLines. The code is more complex and longer.Result Using ReadLine with StreamReader is faster. For files with many lines, it is worth reading in the lines iteratively.using System; using System.Diagnostics; using System.IO; class Program { static void CreateFileWithManyLines() { // Create temporary file for benchmark. using (StreamWriter writer = new StreamWriter(@"C:\programs\file.txt")) { for (int i = 0; i < 1000000; i++) { writer.WriteLine("x"); } } } const int _max = 10; static void Main() { CreateFileWithManyLines(); // Version 1: use StreamReader and read in each line. var s1 = Stopwatch.StartNew(); for (int i = 0; i < _max; i++) { if (Method1() == 0) { return; } } s1.Stop(); // Version 2: use File.ReadAllLines to get entire string array. var s2 = Stopwatch.StartNew(); for (int i = 0; i < _max; i++) { if (Method2() == 0) { return; } } s2.Stop(); Console.WriteLine(s1.Elapsed.TotalMilliseconds.ToString("0.00 ms")); Console.WriteLine(s2.Elapsed.TotalMilliseconds.ToString("0.00 ms")); } static int Method1() { int count = 0; using (StreamReader reader = new StreamReader(@"C:\programs\file.txt")) { while (true) { string line = reader.ReadLine(); if (line == null) { break; } count++; } } return count; } static int Method2() { string[] array = File.ReadAllLines(@"C:\programs\file.txt"); return array.Length; } }219.53 ms StreamReader, ReadLine 1212.43 ms File.ReadAllLinesEven with the helpful types provided in .NET, file handling involves many errors. We must account for disk problems and invalid data.