

# File path formats on Windows systems

Article • 08/20/2022 • 15 minutes to read

Members of many of the types in the [System.IO](#) namespace include a `path` parameter that lets you specify an absolute or relative path to a file system resource. This path is then passed to [Windows file system APIs](#). This topic discusses the formats for file paths that you can use on Windows systems.

## Traditional DOS paths

A standard DOS path can consist of three components:

- A volume or drive letter followed by the volume separator ( `:` ).
- A directory name. The [directory separator character](#) separates subdirectories within the nested directory hierarchy.
- An optional filename. The [directory separator character](#) separates the file path and the filename.

If all three components are present, the path is absolute. If no volume or drive letter is specified and the directory name begins with the [directory separator character](#), the path is relative from the root of the current drive. Otherwise, the path is relative to the current directory. The following table shows some possible directory and file paths.

Path	Description
C:\Documents\Newsletters\Summer2018.pdf	An absolute file path from the root of drive <code>c:</code> .
\Program Files\Custom Utilities\StringFinder.exe	An absolute path from the root of the current drive.
2018\January.xlsx	A relative path to a file in a subdirectory of the current directory.
..\Publications\TravelBrochure.pdf	A relative path to a file in a directory starting from the current directory.
C:\Projects\apilibrary\apilibrary.sln	An absolute path to a file from the root of drive <code>c:</code> .
C:Projects\apilibrary\apilibrary.sln	A relative path from the current directory of the <code>c:</code> drive.

### Important

Note the difference between the last two paths. Both specify the optional volume specifier (`c:` in both cases), but the first begins with the root of the specified volume, whereas the second does not. As result, the first is an absolute path from the root directory of drive `c:`, whereas the second is a relative path from the current directory of drive `c:`. Use of the

second form when the first is intended is a common source of bugs that involve Windows file paths.

You can determine whether a file path is fully qualified (that is, if the path is independent of the current directory and does not change when the current directory changes) by calling the [Path.IsPathFullyQualified](#) method. Note that such a path can include relative directory segments ( . and .. ) and still be fully qualified if the resolved path always points to the same location.

The following example illustrates the difference between absolute and relative paths. It assumes that the directory D:\FY2018\ exists, and that you haven't set any current directory for D:\ from the command prompt before running the example.

C#

```
using System;
using System.Diagnostics;
using System.IO;
using System.Reflection;

public class Example
{
    public static void Main(string[] args)
    {
        Console.WriteLine($"Current directory is '{Environment.CurrentDirectory}'");
        Console.WriteLine("Setting current directory to 'C:\\'");

        Directory.SetCurrentDirectory(@"C:\");
        string path = Path.GetFullPath(@"D:\FY2018");
        Console.WriteLine($"'D:\\FY2018' resolves to {path}");
        path = Path.GetFullPath(@"D:FY2018");
        Console.WriteLine($"'D:FY2018' resolves to {path}");

        Console.WriteLine("Setting current directory to 'D:\\Docs'");
        Directory.SetCurrentDirectory(@"D:\Docs");

        path = Path.GetFullPath(@"D:\FY2018");
        Console.WriteLine($"'D:\\FY2018' resolves to {path}");
        path = Path.GetFullPath(@"D:FY2018");

        // This will be "D:\Docs\FY2018" as it happens to match the drive of the current
        // directory
        Console.WriteLine($"'D:FY2018' resolves to {path}");

        Console.WriteLine("Setting current directory to 'C:\\'");
        Directory.SetCurrentDirectory(@"C:\");

        path = Path.GetFullPath(@"D:\FY2018");
        Console.WriteLine($"'D:\\FY2018' resolves to {path}");

        // This will be either "D:\FY2018" or "D:\FY2018\FY2018" in the subprocess. In
        // the sub process,
        // the command prompt set the current directory before launch of our applica-
        // tion, which
        // sets a hidden environment variable that is considered.
```

```

path = Path.GetFullPath(@"D:FY2018");
Console.WriteLine($"'D:FY2018' resolves to {path}");

if (args.Length < 1)
{
    Console.WriteLine(@"Launching again, after setting current directory to
D:\FY2018");
    Uri currentExe = new Uri(Assembly.GetExecutingAssembly().GetName().CodeBase,
UriKind.Absolute);
    string commandLine = $"/C cd D:\\FY2018 & \"{currentExe.LocalPath}\" stop";
    ProcessStartInfo psi = new ProcessStartInfo("cmd", commandLine); ;
    Process.Start(psi).WaitForExit();

    Console.WriteLine("Sub process returned:");
    path = Path.GetFullPath(@"D:\FY2018");
    Console.WriteLine($"'D:\\FY2018' resolves to {path}");
    path = Path.GetFullPath(@"D:FY2018");
    Console.WriteLine($"'D:FY2018' resolves to {path}");
}
Console.WriteLine("Press any key to continue... ");
Console.ReadKey();
}
}

// The example displays the following output:
//     Current directory is 'C:\Programs\file-paths'
//     Setting current directory to 'C:\'
//     'D:\FY2018' resolves to D:\FY2018
//     'D:FY2018' resolves to d:\FY2018
//     Setting current directory to 'D:\Docs'
//     'D:\FY2018' resolves to D:\FY2018
//     'D:FY2018' resolves to D:\Docs\FY2018
//     Setting current directory to 'C:\'
//     'D:\FY2018' resolves to D:\FY2018
//     'D:FY2018' resolves to d:\FY2018
//     Launching again, after setting current directory to D:\FY2018
//     Sub process returned:
//     'D:\FY2018' resolves to D:\FY2018
//     'D:FY2018' resolves to d:\FY2018
// The subprocess displays the following output:
//     Current directory is 'C:\'
//     Setting current directory to 'C:\'
//     'D:\FY2018' resolves to D:\FY2018
//     'D:FY2018' resolves to D:\FY2018\FY2018
//     Setting current directory to 'D:\Docs'
//     'D:\FY2018' resolves to D:\FY2018
//     'D:FY2018' resolves to D:\Docs\FY2018
//     Setting current directory to 'C:\'
//     'D:\FY2018' resolves to D:\FY2018
//     'D:FY2018' resolves to D:\FY2018\FY2018

```

If you would like to see code comments translated to languages other than English, let us know in [this GitHub discussion issue](#) .

## UNC paths

Universal naming convention (UNC) paths, which are used to access network resources, have the following format:

- A server or host name, which is prefaced by `\\`. The server name can be a NetBIOS machine name or an IP/FQDN address (IPv4 as well as v6 are supported).
- A share name, which is separated from the host name by `\`. Together, the server and share name make up the volume.
- A directory name. The [directory separator character](#) separates subdirectories within the nested directory hierarchy.
- An optional filename. The [directory separator character](#) separates the file path and the filename.

The following are some examples of UNC paths:

Path	Description
<code>\\system07\C\$\</code>	The root directory of the <code>c:</code> drive on <code>system07</code> .
<code>\\Server2\Share\Test\Foo.txt</code>	The <code>Foo.txt</code> file in the <code>Test</code> directory of the <code>\\Server2\Share</code> volume.

UNC paths must always be fully qualified. They can include relative directory segments (`.` and `..`), but these must be part of a fully qualified path. You can use relative paths only by mapping a UNC path to a drive letter.


## DOS device paths

The Windows operating system has a unified object model that points to all resources, including files. These object paths are accessible from the console window and are exposed to the Win32 layer through a special folder of symbolic links that legacy DOS and UNC paths are mapped to. This special folder is accessed via the DOS device path syntax, which is one of:

```
\\.\C:\Test\Foo.txt  \\?\C:\Test\Foo.txt
```

In addition to identifying a drive by its drive letter, you can identify a volume by using its volume GUID. This takes the form:

```
\\.\Volume{b75e2c83-0000-0000-0000-602f00000000}\Test\Foo.txt  \\?\Volume{b75e2c83-0000-0000-0000-602f00000000}\Test\Foo.txt
```

 **Note**

DOS device path syntax is supported on .NET implementations running on Windows starting with .NET Core 1.1 and .NET Framework 4.6.2.

The DOS device path consists of the following components:

- The device path specifier (`\\.\` or `\\?\`), which identifies the path as a DOS device path.

#### ❗ Note

The `\\?\` is supported in all versions of .NET Core and .NET 5+ and in .NET Framework starting with version 4.6.2.

- A symbolic link to the "real" device object (C: in the case of a drive name, or Volume{b75e2c83-0000-0000-0000-602f00000000} in the case of a volume GUID).

The first segment of the DOS device path after the device path specifier identifies the volume or drive. (For example, `\\?\C:\` and `\\.\BootPartition\.`)

There is a specific link for UNC paths that is called, not surprisingly, `UNC`. For example:

```
\\.\UNC\Server\Share\Test\Foo.txt  \\?\UNC\Server\Share\Test\Foo.txt
```

For device UNC paths, the server/share portion forms the volume. For example, in `\\?\server1\share\utilities\filecomparer\`, the server/share portion is `server1\utilities`. This is significant when calling a method such as [Path.GetFullPath\(String, String\)](#) with relative directory segments; it is never possible to navigate past the volume.

DOS device paths are fully qualified by definition and cannot begin with a relative directory segment (`.` or `..`). Current directories never enter into their usage.

## Example: Ways to refer to the same file

The following example illustrates some of the ways in which you can refer to a file when using the APIs in the [System.IO](#) namespace. The example instantiates a [FileInfo](#) object and uses its [Name](#) and [Length](#) properties to display the filename and the length of the file.

C#

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        string[] filenames = {
            @"c:\temp\test-file.txt",
            @"\\127.0.0.1\c$\temp\test-file.txt",
            @"\\localhost\c$\temp\test-file.txt",
            @"\\.\c:\temp\test-file.txt",
            @"\\?\c:\temp\test-file.txt",
            @"\\.\UNC\localhost\c$\temp\test-file.txt",
            @"\\127.0.0.1\c$\temp\test-file.txt" };
    }
}
```

```

foreach (var filename in filenames)
{
    FileInfo fi = new FileInfo(filename);
    Console.WriteLine($"file {fi.Name}: {fi.Length:N0} bytes");
}
}
// The example displays output like the following:
//     file test-file.txt: 22 bytes
//     file test-file.txt: 22 bytes
//     file test-file.txt: 22 bytes
//     file test-file.txt: 22 bytes
//     file test-file.txt: 22 bytes
//     file test-file.txt: 22 bytes
//     file test-file.txt: 22 bytes

```

## Path normalization

Almost all paths passed to Windows APIs are normalized. During normalization, Windows performs the following steps:

- Identifies the path.
- Applies the current directory to partially qualified (relative) paths.
- Canonicalizes component and directory separators.
- Evaluates relative directory components ( . for the current directory and .. for the parent directory).
- Trims certain characters.

This normalization happens implicitly, but you can do it explicitly by calling the [Path.GetFullPath](#) method, which wraps a call to the [GetFullPathName\(\)](#) function. You can also call the Windows [GetFullPathName\(\)](#) function directly using P/Invoke.

## Identify the path

The first step in path normalization is identifying the type of path. Paths fall into one of a few categories:

- They are device paths; that is, they begin with two separators and a question mark or period (\\? or \\.).
- They are UNC paths; that is, they begin with two separators without a question mark or period.
- They are fully qualified DOS paths; that is, they begin with a drive letter, a volume separator, and a component separator (c:\).
- They designate a legacy device (CON, LPT1).
- They are relative to the root of the current drive; that is, they begin with a single component separator (\).

- They are relative to the current directory of a specified drive; that is, they begin with a drive letter, a volume separator, and no component separator (c:).
- They are relative to the current directory; that is, they begin with anything else (temp\testfile.txt).

The type of the path determines whether or not a current directory is applied in some way. It also determines what the "root" of the path is.

## Handle legacy devices

If the path is a legacy DOS device such as CON, COM1, or LPT1, it is converted into a device path by prepending \\.\ and returned.

A path that begins with a legacy device name is always interpreted as a legacy device by the [Path.GetFullPath\(String\)](#) method. For example, the DOS device path for CON.TXT is \\.\CON, and the DOS device path for COM1.TXT\file1.txt is \\.\COM1.

## Apply the current directory

If a path isn't fully qualified, Windows applies the current directory to it. UNC and device paths do not have the current directory applied. Neither does a full drive with separator c:\.

If the path starts with a single component separator, the drive from the current directory is applied. For example, if the file path is \utilities and the current directory is C:\temp\, normalization produces C:\utilities.

If the path starts with a drive letter, volume separator, and no component separator, the last current directory set from the command shell for the specified drive is applied. If the last current directory was not set, the drive alone is applied. For example, if the file path is D:sources, the current directory is C:\Documents\, and the last current directory on drive D: was D:\sources\, the result is D:\sources\sources. These "drive relative" paths are a common source of program and script logic errors. Assuming that a path beginning with a letter and a colon isn't relative is obviously not correct.

If the path starts with something other than a separator, the current drive and current directory are applied. For example, if the path is filecompare and the current directory is C:\utilities\, the result is C:\utilities\filecompare\.

### Important

Relative paths are dangerous in multithreaded applications (that is, most applications) because the current directory is a per-process setting. Any thread can change the current directory at any time. Starting with .NET Core 2.1, you can call the [Path.GetFullPath\(String\)](#),



**String**) method to get an absolute path from a relative path and the base path (the current directory) that you want to resolve it against.

## Canonicalize separators

All forward slashes (/) are converted into the standard Windows separator, the back slash (\). If they are present, a series of slashes that follow the first two slashes are collapsed into a single slash.

## Evaluate relative components

As the path is processed, any components or segments that are composed of a single or a double period ( . or .. ) are evaluated:

- For a single period, the current segment is removed, since it refers to the current directory.
- For a double period, the current segment and the parent segment are removed, since the double period refers to the parent directory.

Parent directories are only removed if they aren't past the root of the path. The root of the path depends on the type of path. It is the drive (c:\) for DOS paths, the server/share for UNC's (\\Server\Share), and the device path prefix for device paths (\\?\ or \\.\).

## Trim characters

Along with the runs of separators and relative segments removed earlier, some additional characters are removed during normalization:

- If a segment ends in a single period, that period is removed. (A segment of a single or double period is normalized in the previous step. A segment of three or more periods is not normalized and is actually a valid file/directory name.)
- If the path doesn't end in a separator, all trailing periods and spaces (U+0020) are removed. If the last segment is simply a single or double period, it falls under the relative components rule above.

This rule means that you can create a directory name with a trailing space by adding a trailing separator after the space.

### 📘 Important

You should **never** create a directory or filename with a trailing space. Trailing spaces can make it difficult or impossible to access a directory, and applications commonly fail when attempting to handle directories or files whose names include trailing spaces.



# Skip normalization

Normally, any path passed to a Windows API is (effectively) passed to the [GetFullPathName function](#) and normalized. There is one important exception: a device path that begins with a question mark instead of a period. Unless the path starts exactly with `\\?\` (note the use of the canonical backslash), it is normalized.

Why would you want to skip normalization? There are three major reasons:

1. To get access to paths that are normally unavailable but are legal. A file or directory called `hidden.`, for example, is impossible to access in any other way.
2. To improve performance by skipping normalization if you've already normalized.
3. On .NET Framework only, to skip the `MAX_PATH` check for path length to allow for paths that are greater than 259 characters. Most APIs allow this, with some exceptions.

## ⓘ Note

.NET Core and .NET 5+ handles long paths implicitly and does not perform a `MAX_PATH` check. The `MAX_PATH` check applies only to .NET Framework.

Skipping normalization and max path checks is the only difference between the two device path syntaxes; they are otherwise identical. Be careful with skipping normalization, since you can easily create paths that are difficult for "normal" applications to deal with.

Paths that start with `\\?\` are still normalized if you explicitly pass them to the [GetFullPathName function](#).

You can pass paths of more than `MAX_PATH` characters to [GetFullPathName](#) without `\\?\`. It supports arbitrary length paths up to the maximum string size that Windows can handle.

## Case and the Windows file system

A peculiarity of the Windows file system that non-Windows users and developers find confusing is that path and directory names are case-insensitive. That is, directory and file names reflect the casing of the strings used when they are created. For example, the method call

```
C#
```

```
Directory.Create("TeStDiReCtOrY");
```

creates a directory named `TeStDiReCtOrY`. If you rename a directory or file to change its case, the directory or file name reflects the case of the string used when you rename it. For example, the following code renames a file named `test.txt` to `Test.txt`:

C#

```
using System.IO;

class Example
{
    static void Main()
    {
        var fi = new FileInfo(@".\test.txt");
        fi.MoveTo(@".\Test.txt");
    }
}
```

However, directory and file name comparisons are case-insensitive. If you search for a file named "test.txt", .NET file system APIs ignore case in the comparison. "Test.txt", "TEST.TXT", "test.TXT", and any other combination of uppercase and lowercase letters will match "test.txt".