

Tema 8.1

Descargar estos apuntes en [pdf](#) o [html](#)

Índice

1. [Definiciones básicas](#)
2. [Gestión de rutas en .NET](#)
 1. [Clase Path](#)
 1. [Operaciones con rutas](#)
3. [Gestión de archivos y directorios](#)
 1. [Clases de utilidad File y Directory](#)
 1. [Ejemplo de uso de File y Directory](#)
 2. [Clases FileInfo y DirectoryInfo](#)
 1. [Ejemplos de uso de FileInfo y DirectoryInfo](#)
 2. [🎓 Caso de estudio de gestión de archivos y directorios](#)

Definiciones básicas

El siguiente '*pre-conocimiento*' será necesario tenerlo presente para abordar el siguiente tema. Por pertenecer los siguiente conceptos al currículo del módulo de '**Sistemas Informáticos**', únicamente vamos a enumerar los conceptos que vamos a necesitar.

- Un **archivo o fichero** informático es un conjunto de bytes que son almacenados en un dispositivo.
(Fuente Wikipedia)
- En informática, **una ruta (path, en inglés)** es la forma de referenciar un archivo informático o directorio en un sistema de archivos de un sistema operativo determinado.
(Fuente Wikipedia)
- Las **rutas absolutas** señalan la ubicación de un archivo o directorio desde el **directorio raíz** del sistema de archivos.
(Fuente Wikipedia)
- Las **rutas relativas** señalan la ubicación de un archivo o directorio a partir de nuestra **posición actual** en el sistema de archivos.
(Fuente Wikipedia)

Gestión de rutas en .NET

Al tratarse .NET de un Framework **multiplataforma**, dispondrá de una serie de **clases de utilidad para el manejo de rutas** de forma '*transparente*' al **Sistema Operativo (SO)** en el que estemos ejecutando.

Clase Path

Esta clase nos proporcionará de forma abstracta las siguientes propiedades, que pueden tomar valores diferentes dependiendo del SO donde estemos ejecutando.

Campo	Parte de la ruta que su valor representa
<code>DirectorySeparatorChar</code>	Separador de directorios. En Windows es <code>\</code> , en Unix es <code>/</code> y Macintosh es <code>:</code>
<code>AltDirectorySeparatorChar</code>	Carácter alternativo usable como separador de directorios. En Windows y Macintosh es <code>/</code> , - mientras que en Unix es <code>\</code>

Campo	Parte de la ruta que su valor representa
<code>PathSeparator</code>	Separador entre rutas. Aunque en los sistemas operativos más comunes es ; podría variar en otros.
<code>VolumeSeparatorChar</code>	Separador de unidades lógicas. En Windows y Macintosh es : (por ejemplo <code>C:\datos</code>) y en Unix <code>/</code>

Ejemplo: Si en lugar de asignar el siguiente literal en el código ...

```
string ruta = @"\datos\fichero.txt";
```

escribimos...

```
char s = Path.DirectorySeparatorChar;
string ruta = $"{s}datos{s}fichero.txt";
```

Conseguiremos que la variable `ruta` almacene el formato de la misma según corresponda al sistema operativo sobre el que se ejecute el código anterior. Es decir, mientras que en **Windows** contendría `\datos\fichero.txt` , en **Linux o Mac OSX** contendría `/datos/fichero.txt`

Operaciones con rutas

Si te fijas en el ejemplo anterior, ocurre que en Windows el carácter usado como separador de directorios `\` coincide con el que C# usa como indicador de secuencias de escape. Por eso es incorrecto indicar literales de cadena para rutas como `"C:\datos"` , ya que C# entendería que estamos intentando escapar el carácter `d` .

En su lugar hay tres alternativas:

1. Usar el campo independiente del sistema operativo `$"C:{Path.DirectorySeparatorChar}datos"`
2. Duplicar los caracteres de los literales para que dejen de considerarse secuencias de escape. Así, la ruta de ejemplo anterior quedaría... `"C:\\datos"`
3. Lo que hemos hecho en nuestro ejemplo que consiste en especificar la ruta mediante un literal de cadena plano, pues en ellos no se tienen en cuenta las secuencias de escape. Así, ahora la ruta del ejemplo quedaría ... como `@"C:\datos"` .

Esta opción es la más simple, si sabemos cual es el SO donde vamos a ejecutar.


- Obtener la ruta con el **directorio 'padre'** o **null** si es raíz.

string Path.GetDirectoryName(string path)

- `GetDirectoryName(@"C:\MyDir\MySubDir\myfile.ext")` devuelve `"C:\MyDir\MySubDir"`
 - `GetDirectoryName(@"C:\MyDir\MySubDir")` devuelve `"C:\MyDir"`
 - `GetDirectoryName(@"C:\MyDir\")` devuelve `"C:\MyDir"`
 - `GetDirectoryName(@"C:\MyDir")` devuelve `"C:\"`
 - `GetDirectoryName(@"C:\")` devuelve `null`
- Obtener el archivo o directorio del final de la ruta. (vacío si acaba en separador)

string Path.GetFileName(string path)

- `GetFileName(@"C:\MyDir\MySubDir\myfile.ext")` devuelve `"myfile.ext"`
 - `GetFileName(@"C:\MyDir\MySubDir")` devuelve `"MySubDir"`
 - `GetFileName(@"C:\MyDir\")` devuelve `""` (Cadena vacía)
- Combinar un **DirectoryName path1** y un **FileName path2** para formar **una nueva ruta**.

 **Nota:** Siempre que **path2** no empiece por el carácter separador de directorio o de volumen.

string Combine(string path1, string path2);

- `Combine(@"C:\MyDir", @"myfile.ext")` devuelve `"C:\MyDir\myfile.ext"`
- `Combine(@"C:\MyDir\", @"myfile.ext")` devuelve `"C:\MyDir\myfile.ext"`
- `Combine(@"C:\MyDir", @"MySubDir\myfile.ext")` devuelve `"C:\MyDir\MySubDir\myfile.ext"`
- `Combine(@"C:\MyDir\", @"MySubDir\myfile.ext")` devuelve `"C:\MyDir\MySubDir\myfile.ext"`
-  `Combine(@"C:\MyDir\", @"\MySubDir\myfile.ext")`
-  `Combine(@"C:\MyDir\", @"C:\MySubDir\myfile.ext")`

Gestión de archivos y directorios

Clases de utilidad File y Directory

Ambas clases contienen gran cantidad de **métodos estáticos de utilidad** para hacer **operaciones** con ficheros/archivos y directorios/carpetas, del estilo de las que se hacen desde la **línea de comandos**.

Lógicamente los métodos definidos en **Directory** realizan operaciones sobre **directorios** y los definidos en **File** sobre **archivos**.

Nota: Como no es idea de estos temas copiar la documentación de Microsoft en castellano. Es recomendable que le eches un vistazo a los enlaces del anterior párrafo para hacerte una idea de las operaciones definidas en estas clases.

Ejemplo de uso de File y Directory

Veamos alguno de estos métodos a través del siguiente **ejemplo comentado**:

```
static void Main()
{
    char s = Path.DirectorySeparatorChar;
    string ruta = $"{s}datos{s}datos.txt";

    // Si no existe el directorio datos en la ruta relativa actual lo crearé.
    if (Directory.Exists(Path.GetDirectoryName(ruta)) == false)
        Directory.CreateDirectory("datos");

    // Creo el fichero datos.txt vacío. Más adelante en el tema
    // veremos que llamar al Close() es importante para que no
    // se quede abierto.
    File.Create(ruta).Close();

    // Me sitúo en el directorio datos.
    Directory.SetCurrentDirectory(Path.GetDirectoryName(ruta) ?? $"{s}");

    Console.WriteLine("El fichero " + ruta + " ");

    // Compruebo si se ha creado el fichero correctamente viendo si
    // existe o no en el directorio datos (donde me acabo de situar).
    // Mostraré si existe o no por pantalla.
    Console.WriteLine(File.Exists(Path.GetFileName(ruta)) ? "existe" : "no existe");
}
```

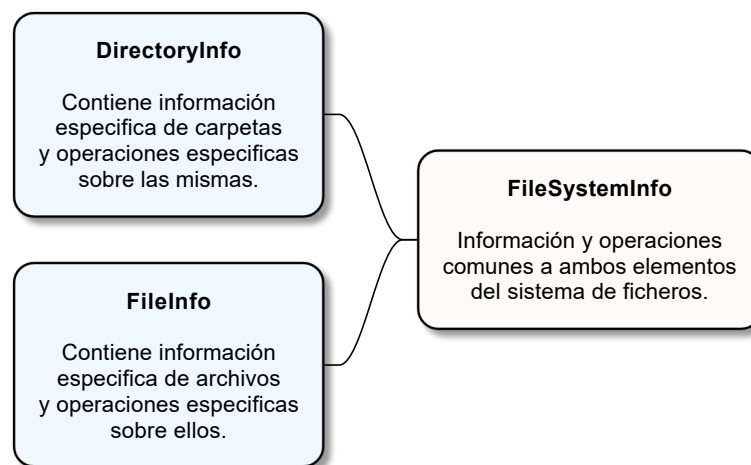
Clases FileInfo y DirectoryInfo

Además de las clases con métodos de operaciones sobre directorios y ficheros. También dispondremos de objetos que contendrán información sobre los mismos además de permitirnos también hacer ciertas operaciones.

Dichos objetos se instanciarán en memoria a través de las clases...

DirectoryInfo y **FileInfo** pues ...

- Me devolverán información sobre **carpetas** y **archivos** respectivamente.
- Además de permitirme '*navegar*' por el sistema de ficheros, moviéndome entre las diferentes carpetas. Me permitirán **realizar acciones más precisas y con más detalle** sobre el sistema de ficheros.
- **Ambas clases heredarán de la clases abstracta** `FileSystemInfo` que contendrá la información común a archivos y carpetas. Un ejemplo sería la fecha de creación y otras propiedades comunes.
- La abstracción del sistema de ficheros se almacena en la clase `FileSystemInfo` :
 - Se accederá a través de la sustitución de Liskov de objetos de tipo `DirectoryInfo` o `FileInfo` .



- Un ejemplo de esta información común detallada. Es el campo `Attributes` del sistema de ficheros.

El campo `Attributes` es una máscara creada a partir del Enum `FileAttributes`. Con información común a ficheros y directorios.

Nota: Échale un vistazo a los bits de la máscara en el enlace del tipo.

Aunque sería inabordable tratar todos los casos de uso de estos objetos. Vamos a ver un par de ejemplos y un caso de uso.

Ejemplos de uso de FileInfo y DirectoryInfo

Ejemplo 1:

Supongamos una función que recibe la ruta al fichero del sistema Windows 10

`C:\Windows\write.exe` y nos muestra la información común almacenada en la clase abstracta `FileSystemInfo` sobre él.

```
class EjemploFileInfo
{
    static string ObtenInformacion(string rutaAFichero)
    {
        // Obtenemos la información del fichero y
        // hacemos una sustitución a la superclase.
        7  FileSystemInfo f = new FileInfo(rutaAFichero);

        StringBuilder informacion = new StringBuilder();

        // Añadimos información del fichero.
        // Fíjate que f.Attributes muestra todos
        // los valores de enum añadidos a la máscara.
        if (f.Exists)
        15     informacion.Append($"Nombre completo: {f.FullName}\n")
                    .Append($"Nombre : {f.Name}\n")
                    .Append($"Extensión : {f.Extension}\n")
                    .Append($"Fecha creación: {f.CreationTime}\n")
                    .Append($"Fecha último acceso: {f.LastAccessTime}\n")
                    .Append($"Fecha última modificación: {f.LastWriteTime}\n")
        21     .Append($"Atributos: {f.Attributes}\n");
        else
            informacion.Append("Archivo no encontrado");

        return informacion.ToString();
    }
    static void Main()
    {
        Console.WriteLine(ObtenInformacion(@"C:\Windows\write.exe"));
    }
}
```

Ejemplo 2:

Supongamos un programa que me liste lo que contiene el directorio '*HOME*' de un usuario para diferentes SO, indicándome si lo encontrado es un archivo o una carpeta.

```
static void Main()
{
    //La siguiente expresión está fuera del alcance de este curso pero por resumir
    // lo que hace diremos que ...
    // Si me estoy ejecutando en Unix, Linux y MacOS X tomo el valor de la
    // carpeta de usuario de la variable HOME, en caso contrario de la
    // ubicación que indique Windows en su variable de ambiente.
    string home = Environment.OSVersion.Platform == PlatformID.Unix
        ? Environment.GetEnvironmentVariable("HOME") ?? "."
        : Environment.ExpandEnvironmentVariables("%HOMEDRIVE%HOMEPATH%");
    // Compruebo si existe la ruta devuelta por el entorno.
    if (Directory.Exists(home))
    {
        // Me sitúo en el directorio home.
        Directory.SetCurrentDirectory(home);
        // Instancio el objeto de tipo DirectoryInfo con la información de la carpeta.
        17 DirectoryInfo infoCarpeta = new DirectoryInfo(home);

        // Obtengo información de todos los objetos que hay en dicha carpeta ya sean
        // otras carpetas o archivos. Para eso llamo a GetFileSystemInfos() que me
        // devuelve un array de FileSystemInfo con dicha información.
        22 FileSystemInfo[] infosEnFS = infoCarpeta.GetFileSystemInfos();

        // Si hubiera querido ver si hay otras carpetas hubiera hecho...
        25 // -> DirectoryInfo[] infoCarpetas = infoCarpeta.GetDirectories();
        // De forma análoga si hubiera querido coger solo información de archivos...
        27 // -> FileInfo[] infoArchivos = infoCarpeta.GetFiles();

        // Recorro el array.
        foreach (FileSystemInfo infoEnFS in infosEnFS)
        {
            // Compruebo si en la máscara el item que estoy recorriendo
            // me indica que es una carpeta.
            34 bool esCarpeta = (infoEnFS.Attributes & FileAttributes.Directory)
                == FileAttributes.Directory;
            // Muestro el nombre completo indicando si es un archivo o una carpeta.
            string info = $"{(esCarpeta ? "Carpeta":"Archivo")}->{infoEnFS.FullName}";
            Console.WriteLine(info);
        }
    }
    else
        Console.WriteLine("No se ha podido encontrar la carpeta home.");
}
```


Caso de estudio de gestión de archivos y directorios

En el siguiente caso de estudio, vamos a crear un método llamado **void OcultaDirectorio(string ruta)** que reciba una ruta a una carpeta y la marque como oculta.

Además, vamos a controlar las posibles excepciones que se generen y las vamos a relanzar al **Main**. Para ello, si hacemos **Ctrl + Click** sobre el constructor de **DirectoryInfo** podemos ver que genera las excepciones:

- **UnauthorizedAccessException** : Si no tenemos permiso de acceso a la carpeta.
- **ArgumentException** : Si la ruta contiene algún carácter inválido.
- **PathTooLongException** : Si la ruta es demasiado larga para el SO.

Además de las anteriores, generaré yo la excepción **FileNotFoundException** si no existe la ruta que recibe el método por parámetro.

Una **propuesta de solución** podría ser la siguiente ...

```
static void OcultaDirectorio(string ruta)
{
    string log = $"Ocultando el directorio '{ruta}'";
    // Mensaje para la consola de depuración.
    Debug.WriteLine(log);
    try
    {
        FileSystemInfo d = new DirectoryInfo(ruta);
        // Genero la excepción indicando lo que estoy haciendo y además le añado
        // como innerException otra instancia donde indico realmente el error.
        if (!d.Exists)
            throw new FileNotFoundException(log,
                new FileNotFoundException($"El directorio '{ruta}' no existe"));
        // Añado con un OR de bit el atributo Hidden (Oculto) a la máscara de
        // atributos del FileSystemInfo del directorio.
        d.Attributes |= FileAttributes.Hidden;
    }
    catch (UnauthorizedAccessException e)
    {
        throw new UnauthorizedAccessException(log, e);
    }
    catch (ArgumentException e)
    {
        throw new ArgumentException(log, e);
    }
    catch (PathTooLongException e)
    {
        throw new PathTooLongException(log, e);
    }
}
```

Ahora defino un **Main** donde voy a usar el método definido de tal manera que:


1. Primero creo un directorio llamado **'oculto'** donde estoy ejecutando la aplicación.
2. Posteriormente lo ocultaré llamando al método **OcultarDirectorio**.

Además, capturo cualquier excepción que se pueda producir, tanto creando el directorio prueba, como llamando al método para ocultarlo,

```
static void Main()
{
    try
    {
        DirectoryInfo d = Directory.CreateDirectory("oculto");
        // Fíjate que Directory.CreateDirectory(..) devuelve un DirectoryInfo con la
        // información del directorio que acabo de crear y que aprovecho para
        // pasar la información de la ruta completa a OcultarDirectorio(...)
        OcultarDirectorio(d.FullName);
    }
    catch (Exception? e)
    {
        while (e != null) {
            Console.WriteLine(e.Message);
            e = e.InnerException;
        }
    }
}
```

En este caso de estudio, se propone hacer las siguientes modificaciones...

1. Ejecutalo y comprueba si se ha creado un directorio oculto llamado **'oculto'**.
2. Intenta ocultar un directorio inexistente.
3. Revoca todos los permisos a tu usuario para esa carpeta y prueba a ocultarla.

 **Nota:** En el [siguiente enlace](#) puede ver como se quitan los permisos al Administrador en Windows 10.

Intenta hacer lo mismo pero solo para tu usuario.