

# Sistema de Archivos

*Exámen Final de Programación II - Curso 2022*

**NOTA:** Antes de comenzar asegúrese de descompactar el archivo `filesystem.zip` y abrir la solución `filesystem.sln` en su editor. Asegúrese también de que su código compila, y la aplicación de consola ejecuta (debe lanzar una excepción). Recuerde que todo el código a evaluar debe ir en el archivo `Exam.cs`

En este ejercicio crearemos un sistema de archivos virtual, que simulará algunas de las operaciones que se pueden realizar normalmente en un sistema de archivos real, como el sistema de archivos `ext4` en Linux.

Empecemos por algunas definiciones.

## Carpetas y Archivos

Un sistema de archivos se compone de dos tipos de objetos: carpetas y archivos. Tanto los archivos como las carpetas pueden tener un nombre. Los archivos además tienen un tamaño (que usaremos para simular la idea de tener datos almacenados).

Una carpeta es a su vez un contenedor de otras carpetas y archivos. De esta manera el sistema de archivos en general puede verse como un árbol, cuyas hojas son los archivos, y todos los nodos intermedios son carpetas. La raíz de este árbol es la carpeta con nombre `/`.

En su implementación, usted usará la definición de archivo que se encuentra en el proyecto `filesystem`:

```
public interface IFile
{
    int Size { get; }
    string Name { get; }
}
```

La definición de carpeta también tiene un nombre, y algunos métodos adicionales que usted debe implementar que veremos a continuación:

```
public interface IFolder
{
    string Name { get; }
    // ...
    // otros métodos que veremos a continuación
}
```

El primer método que debe implementar es `CreateFile()` que permite simular el hecho de crear un nuevo archivo en una carpeta:

```
public interface IFolder
{
```

```

    // ...
    IFile CreateFile(string name, int size);
    // ...
}

```

Este método devuelve una instancia de `IFile` que representa el archivo recién creado, y que por supuesto debe coincidir en nombre y tamaño con los argumentos pasados.

Otro método similar es `CreateFolder` que crea una subcarpeta dentro de la carpeta correspondiente:

```

public interface IFolder
{
    // ...
    IFolder CreateFolder(string name);
    // ...
}

```

Al intentar crear una carpeta o archivo que ya existe, usted debe **lanzar una excepción**.

Por supuesto, como mismo es posible crear carpetas y archivos, también es posible enumerar los archivos y carpetas existentes, que deben ser devueltos **en orden alfabético**:

```

public interface IFolder
{
    // ...
    IEnumerable<IFile> GetFiles();
    IEnumerable<IFolder> GetFolders();
    // ...
}

```

Finalmente (por ahora), el método `TotalSize` devuelve el tamaño total de todos los archivos contenidos en la carpeta correspondiente y todas sus subcarpetas, recursivamente. Se asume que las carpetas tienen tamaño 0.

```

public interface IFolder
{
    // ...
    int TotalSize();
}

```

Veamos ahora como se integran estas dos interfaces en un sistema de archivos.

## El Sistema de Archivos

El sistema de archivos se define en la interface `IFileSystem`, que usted debe implementar:

```
public interface IFileSystem
{
    // ... métodos que veremos a continuación
}
```

El sistema de archivos provee un punto de entrada para identificar cualquier archivo o carpeta. Para ello se usan los métodos `GetFolder` y `GetFile` que devuelven respectivamente una carpeta o archivo según su **dirección**.

```
public interface IFileSystem
{
    // ...
    IFolder GetFolder(string path);
    IFile GetFile(string path);
    IFileSystem GetRoot(string path);
    // ...
}
```

La dirección de un archivo o carpeta es un **string** que contiene todos los nombres desde la raíz del sistema de archivos hasta el archivo o carpeta correspondiente, usando `/` como separador. Por ejemplo:

- `/` representa la carpeta raíz del sistema de archivos.
- `/folder1` representa la carpeta llamada `folder1` que es una subcarpeta de la carpeta raíz.
- `/folder1/folder2/file1` representa el archivo `file1` que está en la carpeta `folder2`, que es a su vez una subcarpeta de `folder1`, que a su vez es subcarpeta de la raíz.

El método `GetRoot` devuelve un nuevo sistema de archivos cuya raíz es la carpeta pasada como dirección al método. Por lo tanto, es lo mismo decir:

```
var file = fs.GetFile("/folder1/folder2/file.txt");
```

Que decir:

```
var fs2 = fs.GetRoot("/folder1");
var file = fs2.GetFile("/folder2/file.txt");
```

Además de identificar un archivo o carpeta concreto, el sistema de archivos permite también encontrar todos los archivos que cumplan con cierta condición (por ejemplo, que el tamaño sea menor que cierto valor o que el nombre contenga cierto texto). Para ello se usa el predicado siguiente:

```
public delegate bool FileFilter(IFile file);
```

Usando este delegado, el método `Find` enumera todos los archivos que cumplen con dicho predicado, en **preorden**, (primero los archivos de la carpeta actual y luego recursivamente los archivos de las subcarpetas) y recorriendo los archivos y carpetas en orden alfabético.

```
public interface IFileSystem
{
    // ...
    IEnumerable<IFile> Find(FileFilter filter);
    // ...
}
```

Finalmente, el sistema de archivos permite copiar, mover, y eliminar carpetas o archivos, con los métodos siguientes:

```
public interface IFileSystem
{
    // ...
    void Copy(string origin, string destination);
    void Move(string origin, string destination);
    void Delete(string path);
    // ...
}
```

En el caso de `Copy` y `Move`, el origen puede ser un archivo, o una carpeta, y el destino siempre será una carpeta. Si el origen es una carpeta, evidentemente se copiará o moverá la carpeta, subcarpetas, y todos los archivos de forma recursiva. Note que siempre se copia o mueve el origen para *dentro* del destino. O sea que si `origin` apunta a una carpeta, tendremos entonces una nueva subcarpeta dentro de `destination` con todo el contenido correspondiente.

Si al mover una carpeta, ya existe otra carpeta con el mismo nombre en el destino, entonces los contenidos de ambas carpetas **deben mezclarse recursivamente**. En caso de que existan archivos con el mismo nombre, **siempre se reemplazará** el archivo existente por el archivo nuevo.

En el caso de `Delete` el argumento `path` puede ser una carpeta o archivo. En caso de ser una carpeta, se elimina todo su contenido.

La diferencia entre copiar y mover es que al mover, se elimina lo movido de su lugar de origen.

Por razones de seguridad, no es posible copiar, mover, o eliminar la carpeta raíz `/`. Si esto se intenta, usted debe lanzar una excepción.

## Implementando el Sistema de archivos

Evidentemente, usted debe dar una implementación de `IFileSystem`, `IFolder` e `IFile`. Las clases que implementan estas interfaces, y todo el código adicional que haga falta para su funcionamiento, deben estar en el archivo `exam/Exam.cs`, que será **el único archivo evaluado**.

Para evaluar su código, se ejecutará el método `CreateFileSystem` en la clase `Exam`. En este método usted debe simplemente devolver una instancia de su implementación de la interfaz `IFileSystem`.

Recuerde además implementar las propiedades `Nombre` y `Grupo` de la clase `Exam`.

En el proyecto `exam`, archivo `Program.cs`, que es una aplicación de consola, usted puede adicionar todo el código que considere necesario para probar su implementación. Ese código no será evaluado.

Para todos los llamados a todos los métodos que no sean correctos (por ejemplo, pedir una carpeta o archivo que no exista, mover la raíz, o crear un archivo con un nombre ya existente), usted puede lanzar una excepción correspondiente.

## Ejemplo

A continuación mostramos un ejemplo sencillo. El código de este ejemplo está en el método `Main` de la clase `Program`.

Empezamos por instanciar un sistema de archivos nuevo:

```
var fs = Exam.CreateFileSystem();
```

Luego vamos a crear dos carpetas en la raíz. Para ello primero debemos obtener una referencia a la raíz, y luego llamar al método `CreateFolder`:

```
var root = fs.GetFolder("/");
var home = root.CreateFolder("home");
var tmp = root.CreateFolder("tmp");
```

Ahora podemos crear algunos archivos inútiles:

```
for (int i = 0; i < 10; i++)
    tmp.CreateFile($"file{i}.tmp", 10);
```

Dado que hay 10 archivos de tamaño 10, el tamaño total de la carpeta `tmp` debe ser:

```
Debug.Assert(tmp.TotalSize() == 100);
```

Hagamos lo propio con la carpeta `home`, pero esta vez con archivos útiles:

```
home.CreateFile("picture.png", 20);
home.CreateFile("document.docx", 150);
home.CreateFile("virus.exe", 300);
```

Verifiquemos entonces que este archivo tan interesante que acabamos de crear está efectivamente en la carpeta correspondiente:

```
var virusFile = fs.GetFile("/home/virus.exe");
Debug.Assert(virusFile.Name == "virus.exe");
```

Verificamos el método `Find` con dos expresiones lambda, una para identificar archivos grandes, y otras para identificar archivos con un nombre concreto:

```
// Verificando el método `Find` con archivos grandes
foreach (var file in fs.Find(file => file.Size > 50))
    Debug.Assert(file.Size > 50);
```

```
// Verificando el método `Find` con nombres  
foreach (var file in fs.Find(file => file.Name.EndsWith(".png")))  
    Debug.Assert(file.Name == "picture.png");
```

Y finalmente vamos a copiar /tmp para dentro de /home y verificar que efectivamente tenemos dos copias de todos esos archivos, y que los tamaños coinciden:

```
fs.Copy("/tmp", "/home");  
Debug.Assert(home.TotalSize() == 570);  
Debug.Assert(fs.GetFolder("/tmp").TotalSize() ==  
    fs.GetFolder("/home/tmp").TotalSize());
```