Files System Abstraction
Cpt S 422 Homework Assignment
by Evan Olds

Assignment Instructions:

Read all instructions *carefully* before you write any code.

In this assignment you will implement a set of abstract base classes to represent a simple file system. You will also inherit from these classes to implement a disk-based file system and a memory-based file system. In addition, you will generate a set of unit tests for your classes.

Create the File System Classes in FileSystemDefs.cs

Implement the 9 mentioned classes in a single file named FileSystemDefs.cs. You will NOT be implementing your unit tests in this file, so do NOT have a using declaration for NUnit or for any other unneeded namespaces for that matter.

1. Create a public, abstract base class named **Dir422** in the CS422 namespace. This class represents a directory in the file system. Add the following abstract, public properties and functions:

- string Name { get; }
  - A get-only property for the name of the directory. This is NOT the full path, it is just the directory name.
- IList<Dir422> GetDirs()
  - Gets a list of all the directories contained within this one. This is NOT a recursive search. It only returns the directories that are directly inside this one.
- IList<File422>GetFiles()
  - Analogous to GetDirs, but for files instead. Like GetDirs, it is not recursive.
- Dir422 Parent { get; }
  - A get-only property that returns the parent directory of this one. For all directories except the root of the file system, this must be non-null. It must be null only for the root of the file system.
- bool ContainsFile(string fileName, bool recursive)
  - Searches for a file with the specified name within this directory, optionally recursively searching subdirectories if requested. Returns true if the file is found within the scope of the search, false if it is not found.
  - Must reject file names with path characters. So if the file name string contains the **/** or \ path characters, then return false immediately.
- bool ContainsDir(string dirName, bool recursive)
  - Analogous to ContainsFile, only for directories. Like ContainsFile, this must return false if path characters are present in the name.
- Dir422 GetDir(string dirName)

    - This is a non-recursive search, so it only looks for directories immediately within this one.
    - Return null immediately if the directory name string contains path characters.
    - Returns null if there is no directory with the specified name within this one.
    - Returns a non-null Dir422 object if the directory with the specified name exists within this one.
- File422 GetFile(string fileName)
    - Analogous to GetDir, only for a file. Must return null if the file name string contains path characters (**/** or ****).
- File422 CreateFile(string fileName)
    - First validates the file name to ensure that it does not have any invalid characters. For our file systems we consider the only invalid characters to be the path characters (/ and \). If the file name has these characters, or is null or empty, then null is returned.
    - Otherwise, if the name is ok, the file is created with a size of 0 in this directory. Should the file already exist, it is truncated back to a length of 0, erasing all prior content.
    - A File422 object is returned on success, null on failure.
- Dir422 CreateDir(string fileName)
    - First validates the file name to ensure that it does not have any invalid characters. For our file systems we consider the only invalid characters to be the path characters (/ and \). If the file name has these characters, or is null or empty, then null is returned.
    - Unlike CreateFile, this will NOT delete contents if the directory already exists. Instead it will just return the Dir422 object for the existing directory.
    - If the directory does not exist, it is created and returned.
    - Null is returned on failure.

2. Create the **File422** abstract base class in the CS422 namespace. This class represents a file in the file system. Add the following abstract, public properties and functions:

- string Name { get; }
    - A get-only property that gets the name of the file. This is NOT the full path, it is just the file name.
- Stream OpenReadOnly()
    - Opens the file for reading and returns the read-only stream. Returns null on failure.
- Stream OpenReadWrite()
    - Opens the file for reading and writing and returns the read/write stream. Returns null on failure.
- Dir422 Parent { get; }
    - Returns the directory that this file resides in. Must always be non-null.

3. Create the **FileSys422** abstract base class in the CS422 namespace. Add the following public functions:

- Dir422 GetRoot()

    - Gets the root directory for this file system. The parent of the root of any file system must be null.
    - This property is abstract in this class.
- bool Contains(Dir422 directory)
    - Returns true if the directory is contained anywhere inside this file system, false otherwise.
    - Must NOT search through every single directory in the entire file system to determine this. There is a much better way that you should understand after reading the Dir422 class specification.
    - Implement this function within this class so that inheriting classes will not have to provide their own implementation. Mark it as virtual so that they *can* provide an override, but just make sure that you implement it here so that they don't have to.
- bool Contains(File422 file)
    - Returns true if the file is contained anywhere inside this file system, false otherwise.
    - Must NOT search through every single directory in the entire file system to determine this. There is a much better way that you should understand after reading the Dir422 and File422 class specifications.
    - Implement this function within this class so that inheriting classes will not have to provide their own implementation. Mark it as virtual so that they *can* provide an override, but just make sure that you implement it here so that they don't have to.

4. Implement the **StandardFileSystem**, **StdFSDir**, and **StdFSFile** classes in the same file as the abstract base classes (FileSystemDefs.cs). These are classes that inherit from the corresponding abstract base classes previously implemented and represent access to the standard file system provided by the operating system. The StdFSFile objects will be returning FileStream objects when opened for reading and writing.

Most of the functionality in the implementation of these classes is easily achieved by just using the file I/O functions in .NET that are provided through the System.IO.Directory and System.IO.File classes.

Have a static construction function in the StandardFileSystem that allows you to create the file system from an existing folder on disk. Remember that whatever folder is provided serves as the root of the created file system, and will have a null parent. The signature of the function is below. Should it fail, null is returned.

```
public static StandardFileSystem Create(string rootDir)
```

5. Implement the **MemoryFileSystem**, **MemFSDir**, and **MemFSFile** classes in the same file as the previous classes (FileSystemDefs.cs). These are classes that represent a file system held entirely in memory. Have a public, zero-parameter constructor for MemoryFileSystem to create a new and empty file system.

While much of the functionality in these classes is simple, given that you're just storing everything in memory, there is some complexity in the implementation of the OpenReadOnly and OpenReadWrite functionality. Since we are simulating a file in a file system, we want the functionality to be very close to that of the FileStream. When you open a FileStream for reading only, you can generally open the same file any number of additional times for reading. In other words, you can have an unlimited number of FileStreams open concurrently for a single file, provided they all have read-only access. In contrast, you can (if using the default functionality and not requesting special shared access) have only a single FileStream open for a file with write access. This implies that your stream-oriented opening functions in the MemFSFile implementation must satisfy the following:

- If a stream is open for writing (returned from the OpenReadWrite function), then all calls to OpenReadWrite and OpenReadOnly return null, until that stream gets closed/disposed.
- If no streams are open for writing, then each and every call to OpenReadOnly should succeed and return a new stream that provides read access to the file data.
- OpenReadWrite must fail if there are any streams open already for the file, be they read-only or read/write.

You must also make sure that your MemFSFile class implementation is thread-safe. If multiple threads are trying to open streams concurrently, the above requirements must still hold. The returned streams do not have to be thread safe, just the member functions of the MemFSFile class.

<u>Create unit tests (in other .cs files)</u>

Implement your unit tests either as NUnit tests or as a standalone console application. Make sure the unit tests exercise your implementation against the specification listed above. Also make sure that you're implementing your unit tests in different code files from the file system implementation. Include all relevant files in your submission zip.