Virtual file-system layer

To improve cross platform support, all file access (and path manipulation) is now done through a well known interface (`FileSystem`).

Note that `FileSystem` extends `ReadonlyFileSystem` , which itself extends `PathManipulation` . If you are using a file-system object you should only ask for the type that supports all the methods that you require. For example, if you have a function (`foo()`) that only needs to resolve paths then it should only require `PathManipulation` : `foo(fs: PathManipulation)` . This allows the caller to avoid implementing unneeded functionality.

For testing, a number of `MockFileSystem` implementations are supplied. These provide an in-memory file-system which emulates operating systems like OS/X, Unix and Windows.

The current file system is always available via the helper method, `getFileSystem()` . This is also used by a number of helper methods to avoid having to pass `FileSystem` objects around all the time. The result of this is that one must be careful to ensure that the file-system has been initialized before using any of these helper methods. To prevent this happening accidentally the current file system always starts out as an instance of `InvalidFileSystem` , which will throw an error if any of its methods are called.

Generally it is safer to explicitly pass file-system objects to constructors or free-standing functions if possible. This avoids confusing bugs where the global file-system has not been set-up correctly before calling functions that expect there to be a file-system configured globally.

You can set the current file-system by calling `setFileSystem()` . During testing you can call the helper function `initMockFileSystem(os)` which takes a string name of the OS to emulate, and will also monkey-patch aspects of the TypeScript library to ensure that TS is also using the current file-system.

Finally there is the `NgtscCompilerHost` to be used for any TypeScript compilation, which uses a given file-system.

All tests that interact with the file-system should be tested against each of the mock file-systems. A series of helpers have been provided to support such tests:

- `runInEachFileSystem()` - wrap your tests in this helper to run all the wrapped tests in each of the mock file-systems, it calls `initMockFileSystem()` for each OS to emulate.
- `loadTestFiles()` - use this to add files and their contents to the mock file system for testing.
- `loadStandardTestFiles()` - use this to load a mirror image of files on disk into the in-memory mock file-system.
- `loadFakeCore()` - use this to load a fake version of `@angular/core` into the mock file-system.

All ngcc and ngtsc source and tests now use this virtual file-system setup.