# Chapter 1

# Files with FileSystem

*with the participation of:*
*Max Leske*

Filesystem is the new file system library for Pharo. It is integrated in the system since Pharo 2.0. Filesystem has been originally developed by Colin Putney. Camillo Bruni made some changes to the original design or API and integrated it into Pharo with the help of Esteban Lorenzano and Guillermo Polito - This is this version that we describe in this chapter. This chapter is a quick start that shows how to get started.

## 1.1 Getting started

The framework supports different kinds of filesystems that can be used interchangeably and that can transparently work with each other. The most obvious one is the filesystem on your hard disk. We are going to work with that one for now. FileSystem is the factory to access different filesystem.

Sending the message disk to FileSystem, returns a file system as on your physical hard-drive. Another less used possibility is memory to create a file system at the system of the image.

```
| working |
working := FileSystem disk workingDirectory.
   ⟶   /Users/ducasse/Workspace/FirstCircle//Pharo/20

working := FileSystem disk workingDirectory class
   ⟶   FileReference
```

The message workingDirectory above returns a reference to the working directory. References are instances of the class FileReference. As we will see references are the central objects of the framework and provide the primary mechanisms for working with files and directories.

Filesystem defines four classes that are important for the end-user: FileSystem, FileReference, FileLocator, and FileSystemDirectoryEntry. These classes are grouped in the 'FileSystem-Core-Public' category.

You should do not use platform specific classes such as UnixStore or WindowsStore, these are internal classes. All code snippets below work on FileReference instances.

## 1.2 Navigating the Filesystem

Now let's do some more interesting things. To list the immediate children of your working directory, execute the following expression:

```
| working |
working := FileSystem disk workingDirectory.
working children.
    ⟶    anArray(file:///Users/ducasse/Workspace/FirstCircle/Pharo/20/.DS_Store file:///
        Users/ducasse/Workspace/FirstCircle/Pharo/20/ASAnimation.st ...)
```

Notice that children returns the direct files and folders. To recursively access all the children of the current directory you should use the message allChildren as follows:

```
working allChildren.
```

To find all st files in the working directory, simply execute:

```
working allChildren select: [ :each | each basename endsWith: 'st' ]
```

Use the slash operator to obtain a reference to a specific file or directory within your working directory:

```
| working cache |
working := FileSystem disk workingDirectory.
cache := working / 'package-cache'.
```

Navigating back to the parent is easy using the parent message:

```
| working cache |
working := FileSystem disk workingDirectory.
cache := working / 'package-cache'.
parent := cache parent.
parent = working
    ⟶    true
```

You can check for various properties of the cache directory by executing the following expressions:

```
cache exists.            ⟶   true
cache isSymLink.                ⟶   false
cache isFile.            ⟶   false
cache isDirectory.          ⟶   true
cache basename.           ⟶   'package−cache'
cache fullName            ⟶   '/Users/ducasse/Workspace/FirstCircle/Pharo/20/
    package−cache'
cache parent fullName       ⟶   '/Users/ducasse/Workspace/FirstCircle/Pharo/20/'
```

The methods exists, isFile, isDirectory, and basename are defined on the FileReference class. Notice that there is no message to get the path without the basename and that the idiom is to use parent fullName to obtain it. The message path returns a Path object which is internally used by FileSystem. You normally do not to use such objects.

Note that FileSystem does not really distinguish between files and folders which often leads to cleaner code and can be seen as an application of the composite design patterns.

To get additional information about a filesystem entry, we should get an FileSystemDirectoryEntry using the message entry. Note that you can access the file permissions. Here are some examples:

```
cache entry creation.          ⟶   2012−04−25T15:11:36+02:00
cache entry creationTime        ⟶   2012−04−25T15:11:36+02:00
cache entry creationSeconds      ⟶   3512812296 2012−08−02T14:23:29+02:00
cache entry modificationTime      ⟶   2012−08−02T14:23:29+02:00
cache entry size.          ⟶   0 (directories have size 0)
cache entry permissions        ⟶   rwxr−xr−x
cache entry permissions class         ⟶   FileSystemPermission
cache entry permissions isWritable     ⟶   true
cache entry isFile          ⟶   false
cache entry isDirectory          ⟶   true
```

**Locations.** The framework also supports locations, late-bound references that point to a file or directory. When asking to perform a concrete operation, a location behaves the same way as a reference. Here are some locations.

```
FileLocator desktop.
FileLocator home.
FileLocator imageDirectory.
FileLocator vmDirectory.
```

If you save a location with your image and move the image to a different machine or operating system, a location will still resolve to the expected directory or file. Note that some of them are still in flux because depending on specific VM functionalities.

## 1.3 Opening Read- and Write-Streams

To open a file-stream on a file ask the reference for a read- or write-stream:

```
| working stream |
working := FileSystem disk workingDirectory.
stream := (working / 'foo.txt') writeStream.
stream nextPutAll: 'Hello World'.
stream close.
stream := (working / 'foo.txt') readStream.
stream contents.          ⟶    'Hello World'
stream close.
```

Please note that writeStream overrides any existing file and readStream throws an exception if the file does not exist. There are also short forms available that eliminate the need to close a stream manually:

```
| working  |
working := FileSystem disk workingDirectory.
working / 'foo.txt' writeStreamDo: [ :stream | stream nextPutAll: 'Hello World' ].
working / 'foo.txt' readStreamDo: [ :stream | stream contents ].
```

```
| stream |
stream := FileSystem disk openFileStream: 'authors.txt' writable: true.
stream nextPutAll: 'stephane alexandre damien jannik'.

| working |
working := FileSystem disk workingDirectory.
working / 'authors.txt' readStreamDo: [ :stream | stream contents ].
    ⟶    'stephane alexandre damien jannik'

FileSystem disk workingDirectory / 'authors.txt'
   writeStreamDo: [ :stream | stream nextPutAll: 'andrew oscar'].

| working |
working := FileSystem disk workingDirectory.
working / 'authors.txt' readStreamDo: [ :stream | stream contents ].
     ⟶    'andrew oscarxandre damien jannik'
```

Have a look at the streams protocol of FileReference for other convenience methods.

## 1.4 Renaming, Copying and Deleting Files and Directories

You can also copy and rename files using the methods copyTo: and renameTo:.

```
| working |
working := FileSystem disk workingDirectory.
working / 'foo.txt' writeStreamDo: [ :stream | stream nextPutAll: 'Hello World' ].
working / 'foo.txt' copyTo: (working / 'bar.txt').

| working  |
working := FileSystem disk workingDirectory.
working / 'bar.txt' readStreamDo: [ :stream | stream contents ].
   →     'Hello World'

| working |
working := FileSystem disk workingDirectory.
working / 'foo.txt' renameTo: 'skweek.txt'.

| working  |
working := FileSystem disk workingDirectory.
working / 'skweek.txt' readStreamDo: [ :stream | stream contents ].
   →     'Hello World'
```

To create a directory use the message createDirectory:

```
| working  |
working := FileSystem disk workingDirectory.
backup := working / 'cache−backup'.
backup createDirectory.
backup isDirectory.
      →    true
back children.
      →    #()
```

To then you can copy the contents of the complete package-cache to that directory simply use copyAllTo::

```
cache copyAllTo: backup.
```

Note that the target directory will be automatically created if it was not there before.

To delete a single file, use the message delete:

```
 (working / 'bar.txt') delete.
```

To delete a complete directory tree use deleteAll. Be careful with that one though.

## 1.5 FileReference

FileReference offers a set of operations to manipulate files. We saw some of them until now and here is a more complete list of operations.

**Stéf** ▶*here so far*◀

delete (deletes the file if it is present else raise an error, deleteAll, deleteAllChildren, deleteIfAbsent: ensureDeleted (which makes sure that the file will be deleted),

ensureDirectory does not raise an error while createDirectory does it.

```
(FileSystem disk workingDirectory / 'paf' ) delete.
(FileSystem disk workingDirectory / 'paf' ) createDirectory.
[(FileSystem disk workingDirectory / 'paf' ) createDirectory] on: DirectoryExists do: [:ex|
      true].
      ⟶   true
(FileSystem disk workingDirectory / 'paf' ) delete.
(FileSystem disk workingDirectory / 'paf' ) delete.
(FileSystem disk workingDirectory / 'paf' ) ensureDirectory.
(FileSystem disk workingDirectory / 'paf' ) ensureDirectory.
(FileSystem disk workingDirectory / 'paf' ) isDirectory.
    ⟶   true
```

ensureFile

**Stéf** ▶*what is the difference between deleteAll and deleteAllChildren*◀ **Stéf** ▶*what is the difference between delete and ensureDeleted? since delete sent to a non existing file does not raise error*◀

### Reference

Paths and filesystems are the lowest level of the Filesystem API. A FSReference combines a path and a filesystem into a single object which provides a simpler protocol for working with files. It implements the same operations as FSFilesystem, but without the need to track paths and filesystem separately:

```
fs := FSFilesystem memory.
griffle := fs referenceTo: (FSPath / 'plonk' / 'griffle').
nurp := fs referenceTo: (FSPath * 'nurp').

griffle isFile.          ⟶   false
griffle isDirectory.       ⟶   false
```

```
griffle parent ensureDirectory.
griffle writeStreamDo: [ :stream | ].
griffle exists & griffle isFile    ⟶  true
griffle copyTo: nurp
nurp exists.              ⟶  true
griffle delete
```

References also implement the path protocol with methods like /, parent and resolve:.

## Locator

Locators could be considered late-bound references. They're left deliberately fuzzy, and are only resolved to a concrete reference when some file operation needs to be performed. Instead of a filesystem and path, locators are made up of an origin and a path. An origin is an abstract filesystem location, such as the user's home directory, the image file, or the VM executable. When it receives a message like isFile, a locator will first resolve its origin, then resolve its path against the origin.

Locators make it possible to specify things like "an item named 'package-cache' in the same directory as the image file" and have that specification remain valid even if the image is saved and moved to another directory, possibly on a different computer.

```
locator := FSLocator image / 'package-cache'.
locator printString.        ⟶  '{image}/package-cache'
locator resolve.            ⟶  /Users/colin/Projects/Mason/package-cache
locator isFile.             ⟶  false
locator isDirectory.        ⟶  true
```

The following origins are currently supported:

- imageDirectory - the directory in which the image resides in
- image - the image file
- changes - the changes file
- vmBinary - the executable for the running virtual machine
- vmDirectory - the directory containing the VM application (may not be the parent of vmBinary)
- home - the user's home directory
- desktop - the directory that hold the contents of the user's desktop

- documents - the directory where the user's documents are stored (e.g. '/Users/colin/Documents')

Applications may also define their own origins, but the system will not be able to resolve them automatically. Instead, the user will be asked to manually choose a directory. This choice is then cached so that future resolution requests will not require user interaction.

### Filesystem

A filesystem is an interface to access hierarchies of directories and files. "The filesystem," provided by the host operating system, is represented by FSDiskFilesystem and its platform-specific subclasses. However, the user should not access them directly but using FSFilesystem as we show previously. Other kinds of Filesystems are also possible. The memory filesystem provides a RAM disk filesystem where all files are stored as ByteArrays in the image. The zip filesystem represents the contents of a zip file.

Each filesystem has its own working directory, which it uses to resolve any relative paths that are passed to it. Some examples:

```
fs := FSFilesystem memory.
fs workingDirectoryPath: (FSPath / 'plonk').
griffle := FSPath / 'plonk' / 'griffle'.
nurp := FSPath * 'nurp'.

fs resolve: nurp.                 —→  /plonk/nurp

fs createDirectory: (FSPath / 'plonk').     —→  "/plonk created"
(fs writeStreamOn: griffle) close.     —→  "/plonk/griffle created"
fs isFile: griffle.           —→  true
fs isDirectory: griffle.           —→  false
fs copy: griffle to: nurp.           —→  "/plonk/griffle copied to /plonk/nurp"
fs exists: nurp.             —→  true
fs delete: griffle.          —→  "/plonk/griffle" deleted
fs isFile: griffle.           —→  false
fs isDirectory: griffle.           —→  false
```

## 1.6 Looking at FileSystem internals

Stéf ►*put an uml diagram?*◄ Now we explain the key classes of Filesystem.

## Path

Paths are the most fundamental element of the Filesystem API. They represent filesystem paths in a very abstract sense, and provide a high-level protocol for working with paths without having to manipulate strings. Here are some examples showing how to define absolute paths (/), relative paths (*), file extension (,), parent navigation (parent)

```
"absolute path"
FSPath / 'plonk' / 'feep'          −→  /plonk/feep

"relative path"
FSPath * 'plonk' / 'feep'          −→  plonk/feep

"relative path with extension"
FSPath * 'griffle' , 'txt'         −→  griffle.txt

"changing the extension"
FSPath * 'griffle.txt' , 'jpeg'    −→  griffle.jpeg

"parent directory"
(FSPath / 'plonk' / 'griffle') parent    −→  /plonk

"resolving a relative path"
(FSPath / 'plonk' / 'griffle') resolve: (FSPath * '..' / 'feep')
          −→  /plonk/feep

"resolving an absolute path"
(FSPath / 'plonk' / 'griffle') resolve: (FSPath / 'feep')
          −→  /feep

"resolving a string"
(FSPath * 'griffle') resolve: 'plonk'    −→  griffle/plonk

"comparing"
(FSPath / 'plonk') contains: (FSPath / 'griffle' / 'nurp')
          −→  false
```

Note that some of the path protocol (messages like /, parent and resolve:) are also available on references — references are a combination of path and filesystem.

## Enumeration

References and Locators also provide simple methods for dealing with whole directory trees:

**allChildren.** This will answer an array of references to all the files and directories in the directory tree rooted at the receiver. If the receiver is a file, the array will contain a single reference, equal to the receiver.

**allEntries.** This method is similar to allChildren, but it answers an array of FSDirectoryEntries, rather than references.

**copyAllTo: aReference.** This will perform a deep copy of the receiver, to a location specified by the argument. If the receiver is a file, the file will be copied; if a directory, the directory and its contents will be copied recursively. The argument must be a reference that doesn't exist; it will be created by the copy.

**deleteAll.** This will perform a recursive delete of the receiver. If the receiver is a file, this has the same effect as delete.

## Visitors

The above methods are sufficient for many common tasks, but application developers may find that they need to perform more sophisticated operations on directory trees.

The visitor protocol is very simple. A visitor needs to implement visitFile: and visitDirectory:. The actual traversal of the filesystem is handled by a guide. A guide works with a visitor, crawling the filesystem and notifying the visitor of the files and directories it discovers. There are three Guide classes, FSPreorderGuide, FSPostorderGuide and FSBreadthFirstGuide , which traverse the filesystem in different orders. To arrange for a guide to traverse the filesystem with a particular visitor is simple. Here's an example:

```
FSBreadthFirstGuide show: aReference to: aVisitor
```

The enumeration methods described above are implemented with visitors; see FSCopyVisitor, FSDeleteVisitor, and FSCollectVisitor for examples.