Chapter 1

Files with FileSystem

with the participation of:
Max Leske *(maxleske@gmail.com)*

The library for dealing with files in Pharo is called FileSystem. It offers an expressive and elegant object-oriented design. This chapter presents the key aspects of the API to cover most of the needs one may have.

FileSystem is the result of long and hard work from many people. FileSystem was originally developed by Colin Putney and the library is distributed under the MIT license, as for most components of Pharo. Camillo Bruni made some changes to the original design. Camillo Bruni integrated it into Pharo with the help of Esteban Lorenzano and Guillermo Polito. This chapter would not exist without the previous work of all the contributors of FileSystem. We are grateful to all of them.

1.1 Getting started

The framework supports different "kinds" of filesystems that are interchangeable and may transparently work with each other. The probably most common usage of FileSystem is to directly work with files stored on your hard-drive. We are going to work with that one for now.

The class FileSystem offers factory class-methods to offer access to different filesystems. Sending the message disk to FileSystem, returns a file system as on your physical hard-drive. Sending memory creates a new file system stored in memory 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 directory containing your Pharo image. A reference is an instance of the class FileReference. 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. FileSystem offers factory methods to create a new file system. A FileReference is a reference to a folder or a file and offers methods to navigate and perform operations. A FileLocator is a late binding reference. When asked to perform concrete operation, a file locator looks up the current location of the origin, and resolve the path against it. A FileSystemDirectoryEntry allows one to get the additional information of a file or a directory. These classes belong to the 'FileSystem-Core' package, and are explained below in the chapter.

You should 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 a file system

Now let's play with FileSystem.

Immediate children. 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.
```

Converting a string character into a file reference is a common and handy operation. Simply send asFileReference to a string to get its corresponding file reference.

```
'/Users/ducasse/Workspace/FirstCircle/Pharo/20' asFileReference
```

Note that no error is raised if the string does not point to an existing file. You can however check whether the file exists or not:

```
'foobarzork' asFileReference exists
  ⟶  false
```

All '.st' files. Filtering is realized using standard pattern matching on file name. To find all st files in the working directory, simply execute:

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

The basename message returns the name of the file from a full name (*i.e.*, /foo/gloops.taz basename is 'gloops.taz').

Accessing a given file or directory. 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'.
```

Getting to the parent folder. 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
```

Accessing directory properties. You can check the various properties of an element. For example, in the following we try with the cache directory by executing the following expressions:

```
cache exists.                                  ⟶  true
cache isSymLink. "ask if it is a symbolic link"  ⟶  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 and is not meant to be publicly used.

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 pattern.

Querying file entry status. 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 file locations are specific to the virtual machine.

1.3 Opening read and write Streams

To open a stream on a file, just ask the reference for a read- or write-stream using the message writeStream or readStream as follows:

```
| 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. Forgetting to close stream is a common mistake, for which even advanced programmers regularly fall into. Closing a stream frees low level resources, which is a good thing to do. The messages readStreamDo: and writeStreamDo: frees the programmer from explicitly closing the stream. Consider:

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

Keep in mind that file may be easily overridden without giving any warning. Consider the following situation:

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

The file authors.txt may be simply overridden with:

```
FileSystem disk workingDirectory / 'authors.txt'
  writeStreamDo: [ :stream | stream nextPutAll: 'bob joe'].
```

Reading back the file may give an odd result:

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

We can also use the message openFilestream: aString writable: aBoolean to get a stream with the corresponding write status.

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

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

1.4 Renaming, copying and deleting files and directories

Files may be copied and renamed using the messages copyTo: and renameTo:. Note that while copyTo: tasks as argument another fileReference, renameTo: takes a path, pathname or reference.

```
| 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'
```

Directory creation. To create a directory, use the message createDirectory as follows:

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

Copy everything. You can copy the contents of a directory using the message copyAllTo:. Here we copy the complete package-cache to the backup directory using copyAllTo::

```
cache copyAllTo: backup.
```

Note that before copying the target directory is created if it does not exist.

Deleting. To delete a single file, use the message delete:

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

To delete a complete directory tree (including the receiver) use deleteAll. Be careful to not delete the wrong folder though.

```
backup deleteAll.
```

1.5 The main entry point: FileReference

While FileSystem is based on multiple concepts and classes such as FileSystem, Path and FileReference. FileReference is the most important for an end-user. FileReference offers a set of operations to manipulate files. So far, we have seen some basic operations. This section covers the more elaborated operations.

At the design level, a file reference (FileReference) combines two low-level entities: a path (Path) and a filesystem (FileSystem) into a single object which provides a simple protocol to manipulate and handle files. A FileReference implements many operations of FileSystem (both are largely polymorphic), but without the need to track paths and filesystem separately.

FileReference information access

Once given a file reference you can access usual information using messages basename, base, extensions...

```
| pf |
pf := (FileSystem disk workingDirectory / 'package-cache' ) children second.
    ⟶    /Users/ducasse/Pharo/PharoHarvestingFixes/20/package-cache/AsmJit-
      IgorStasenko.66.mcz
pf fullName
    ⟶    '/Users/ducasse/Pharo/PharoHarvestingFixes/20/package-cache/AsmJit-
      IgorStasenko.66.mcz'
pf basename
    ⟶    'AsmJit-IgorStasenko.66.mcz'
```

```
pf basenameWithoutExtension
    ⟶    'AsmJit−IgorStasenko.66'
pf base
    ⟶    'AsmJit−IgorStasenko'
pf extension
    ⟶    'mcz'
pf extensions
    ⟶    an OrderedCollection('66' 'mcz')
```

Indicators. FileSystem introduces the notion of file reference indicators. An indicator is a visual clue conveying the type of the reference. For now three kind of indicators are implemented, '?' for a non existing reference, '/' for a directory, and the empty string for a file. FileReference defines the message basenameWithIndicator that takes advantage of indicators. The following expressions show its use.

```
pf  basenameWithIndicator
    ⟶    'AsmJit−IgorStasenko.66.mcz'
pf parent  basename
    ⟶    'package−cache'
pf parent basenameWithIndicator
    ⟶    'package−cache/'
```

Path. When there is a need to access a portion of a path, the message pathSegments returns the full name cut into path elements, as strings. Remember that from a design point of view, strings are considered as "dead" objects, so it is often better to deal with the real objects for example using the path message.

```
pf pathSegments
    ⟶    #('Users' 'ducasse' 'Pharo' 'PharoHarvestingFixes' '20' 'package−cache' '
    AsmJit−IgorStasenko.66.mcz')
pf path
    ⟶    Path / 'Users' / 'ducasse' / 'Pharo' / 'PharoHarvestingFixes' / '20' / 'package−
    cache' / 'AsmJit−IgorStasenko.66.mcz'
```

Sizes. FileReference provides also some way to access the size of the file.

```
pf humanReadableSize
    ⟶    '182.78 kB'
pf size
    ⟶    182778
```

File Information. You can get limited information about the file entry itself using creationTime and permissions. To get the full information you should access the entry itself using the message entry.

```
| pf |
pf := (FileSystem disk workingDirectory / 'package-cache' ) children second.
pf creationTime.
   ⟶    2012-06-10T10:43:19+02:00
pf modificationTime.
   ⟶    2012-06-10T10:43:19+02:00
pf permissions
   ⟶    rw-r--r--
```

Entries are objects that represent all the metadata of a single file.

```
| pf |
pf := (FileSystem disk workingDirectory / 'package-cache' ) children second.
pf entry

pf parent entries
   "returns all the entries of the children of the receiver"
```

Operating on files

There are several operations on files.

Deleting. delete, deleteAll, deleteAllChildren, all delete the receiver and raise an error if it does not exist. delete deletes the file, deleteAll deletes the directory and its contents , deleteAllChildren (which only deletes children of a directory). In addition, deleteIfAbsent: executes a block when the file does not exist.

Finally ensureDelete deletes the file but does not raise error if the file does not exist. Similarly ensureDeleteAllChildren, ensureDeleteAll do not raise exception when the receiver does not exist.

```
(FileSystem disk workingDirectory / 'paf') delete.
    ⟶    error
(FileSystem disk workingDirectory / 'fooFolder') deleteAll.
    ⟶    error
(FileSystem disk workingDirectory / 'fooFolder') ensureCreateDirectory.
(FileSystem disk workingDirectory / 'fooFolder') deleteAll.

(FileSystem disk workingDirectory / 'paf') deleteIfAbsent: [Warning signal: 'File did not
      exist'].

(FileSystem disk workingDirectory / 'fooFolder2') deleteAllChildren.
    ⟶    error
```

```
(FileSystem disk workingDirectory / 'fooFolder2') ensureCreateDirectory.
(FileSystem disk workingDirectory / 'fooFolder2') deleteAllChildren.
```

Creating Directory. createDirectory creates a new directory and raises an error if it already exists. ensureCreateDirectory verifies that the directory does not exist and only creates it if necessary. ensureCreateFile creates if necessary a file.

```
(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' ) ensureCreateDirectory.
(FileSystem disk workingDirectory / 'paf' ) ensureCreateDirectory.
(FileSystem disk workingDirectory / 'paf' ) isDirectory.
  ⟶ true
```

Moving/Copying files around. We can move files around using the message moveTo: which expects a file reference.

```
(FileSystem disk workingDirectory / 'targetFolder') exist
  ⟶ false
(FileSystem disk workingDirectory / 'paf') exist
  ⟶ false
(FileSystem disk workingDirectory / 'paf' ) moveTo: (FileSystem disk workingDirectory / '
    targetFolder')
  ⟶ Error

(FileSystem disk workingDirectory / 'paf' ) ensureCreateFile.
(FileSystem disk workingDirectory / 'targetFolder') ensureCreateDirectory.
(FileSystem disk workingDirectory / 'paf' ) moveTo: (FileSystem disk workingDirectory / '
    targetFolder' / 'paf').
(FileSystem disk workingDirectory / 'paf' ) exists.
    ⟶ false
(FileSystem disk workingDirectory / 'targetFolder' / 'paf') exists.
    ⟶ true
```

Besides moving files, we can copy them. We can also use copyAllTo: to copy files. Here, we copy the files contained in the source folder to the target one.

The message copyAllTo: performs a deep copy of the receiver, to a location specified by the argument. If the receiver is a file, the file is copied. If the receiver is a directory, the directory and its contents will be copied recursively.

The argument must be a reference that does not exist; it will be created by the copy.

```
(FileSystem disk workingDirectory / 'sourceFolder') createDirectory.
(FileSystem disk workingDirectory / 'sourceFolder' / 'pif') ensureCreateFile.
(FileSystem disk workingDirectory / 'sourceFolder' / 'paf') ensureCreateFile.
(FileSystem disk workingDirectory / 'targetFolder') createDirectory.
(FileSystem disk workingDirectory / 'sourceFolder') copyAllTo: (FileSystem disk
    workingDirectory / 'targetFolder').
(FileSystem disk workingDirectory / 'targetFolder' / 'pif') exists.
  ⟶  true
(FileSystem disk workingDirectory / 'targetFolder' / 'paf') exists.
  ⟶  true
```

The message copyAllTo: can be used to copy a single file too:

```
(FileSystem disk workingDirectory / 'sourceFolder') ensureCreateDirectory.
(FileSystem disk workingDirectory /  'sourceFolder' / 'pif') ensureCreateFile.
(FileSystem disk workingDirectory /  'sourceFolder' / 'paf') ensureCreateFile.
(FileSystem disk workingDirectory / 'targetFolder') ensureCreateDirectory.
(FileSystem disk workingDirectory / 'sourceFolder' / 'paf') copyAllTo: (FileSystem disk
    workingDirectory / 'targetFolder' / 'paf').
(FileSystem disk workingDirectory /  'targetFolder' / 'paf') exists.
    ⟶  true.
(FileSystem disk workingDirectory /  'targetFolder' / 'pif' ) exists.
  ⟶  false
```

Locator

Locators are late-bound references. They are left deliberately fuzzy, and are only resolved to a concrete reference when some file operation is 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 := FileLocator imageDirectory / 'package-cache'.
locator printString.        ⟶  ' {imageDirectory}/package-cache'
locator resolve.            ⟶  /Users/ducasse/Pharo/PharoHarvestingFixes/20/
    package-cache
locator isFile.             ⟶  false
```

```
locator isDirectory.          ⟶    true
```

The following origins are currently supported:

- imageDirectory - the directory in which the image resides
- 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 holds 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.

absolutePath vs. path. The message absolutePath returns the absolute path of the receiver. When the file reference is not virtual the messages path and absolutePath provide similar results. When the file is a late bound reference (instance of FileLocator), absolutePath resolves the file and returns the absolute path, while path returns an unresolved file reference as shown below.

```
(FileLocator image parent / 'package-cache') path
  ⟶    {image}/../package-cache

(FileLocator image parent / 'package-cache') absolutePath
  ⟶    Path / 'Data' / 'Downloads' / 'Pharo-2.0' / 'package-cache'

(FileLocator image parent / 'package-cache') absolutePath
  ⟶    Path / 'Data' / 'Downloads' / 'Pharo-2.0' / 'package-cache'
```

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

1.6 Looking at FileSystem internals

At that stage, you should be able to comfortably use FileSystem to cover your need file handing. This section is about the internal components of FileSystem. It goes over important implementation details, which will surely interest readers willing to have a new kind of file system, for example on a data base or a remote file system.

FileReference = FileSystem + Path

Paths and filesystems are the lowest level of the FileSystem API. A FileReference combines a path and a filesystem into a single object which provides a simpler protocol for working with files as we show in the previous section. References implement the path protocol with methods like /, parent and resolve:.

FileSystem

A filesystem is an interface to access hierarchies of directories and files. "The filesystem," provided by the host operating system, is represented by DiskStore and its platform-specific subclasses. However, the user should not access them directly but instead use FileSystem as we showed 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 is used to resolve any relative paths that are passed to it. Some examples:

```
fs := FileSystem memory.
fs workingDirectoryPath: (Path / 'plonk').
griffle := Path / 'plonk' / 'griffle'.
nurp := Path * 'nurp'.
fs resolve: nurp.
      ⟶    Path/plonk/nurp

fs createDirectory: (Path / '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
```

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). Normally you do not need to use Path but here are some examples.

```
| fs griffle nurp |
fs := FileSystem memory.
griffle := fs referenceTo: (Path / 'plonk' / 'griffle').
nurp := fs referenceTo: (Path * 'nurp').
griffle isFile.
      ⟶    false
griffle isDirectory.
      ⟶    false
griffle parent ensureCreateDirectory.
griffle ensureCreateFile.
griffle exists & griffle isFile.
      ⟶    true
griffle copyTo: nurp.
nurp exists.
      ⟶    true
griffle delete
```

```
"absolute path"
Path / 'plonk' / 'feep'          ⟶    /plonk/feep

"relative path"
Path * 'plonk' / 'feep'          ⟶    plonk/feep

"relative path with extension"
Path * 'griffle' , 'txt'         ⟶    griffle.txt

"changing the extension"
Path * 'griffle.txt' , 'jpeg'        ⟶    griffle.jpeg

"parent directory"
(Path / 'plonk' / 'griffle') parent      ⟶    /plonk

"resolving a relative path"
(Path / 'plonk' / 'griffle') resolve: (Path * '..' / 'feep')
                    ⟶    /plonk/feep

"resolving an absolute path"
(Path / 'plonk' / 'griffle') resolve: (Path / 'feep')
                    ⟶    /feep
```

```
"resolving a string"
(Path * 'griffle') resolve: 'plonk'      ⟶    griffle/plonk

"comparing"
(Path / 'plonk') contains: (Path / 'griffle' / 'nurp')
                    ⟶    false
```

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

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, PreorderGuide, PostorderGuide and BreadthFirstGuide , 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:

```
BreadthFirstGuide show: aReference to: aVisitor
```

The enumeration methods described above are implemented with visitors; see CopyVisitor, DeleteVisitor, and CollectVisitor for examples.

1.7 Chapter summary

FileSystem is a powerful and elegant library to manipulate files. It is a fundamental part of Pharo. The Pharo community will continue to extend and build it. The class FileReference is the most important entry point to the framework.

- FileSystem offers factory class methods to build file systems on hard disk and in memory.
- FileReference is a central class in the framework which represents a file or a folder. A file reference offers methods to operate on a file and navigate within a file system.
- Sending the message asFileReference to a string character returns its corresponding file reference (*e.g.*, '/tmp' asFileReference)

- Creating a file and writing in it is as simple as: `(FileSystem disk workingDirectory / 'foo.txt') writeStreamDo: [ :stream | stream nextPutAll: 'Hello World' ].`
- `FileLocator` is a late binding reference, useful when the file location in a hard disk depends on the running context.
- `FileSystemDirectoryEntry` offers a large set of low level detail for a given file.