Using FileSystems in Java NIO2

José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard https://github.com/JosePaumard

Agenda

The NIO2 API provides classes to access the file system directly

Implements file systems dependent operations not provided by Java I/O

The FileSystemProvider class

The FileSystem class

The FileStore class

All highly OS / FS dependent!

Introducing File Systems Support

Why add this file system support?

- we already have methods to move files around, create directories

- and get the content of a directory

The answer is: performance

- plugged into the native file system

- can handle very large directories

What Is a File System?

A file system in Java NIO2 is an abstraction of a real file system

It is bound to a scheme (`file://`)

It implements all the operations of the Files class (creation, copy, deletion, etc...)

The JDK provides two file systems: the default one, and a JAR file system

Built on Three Classes

FileSystemProvider acts as a factory for file systems

- can create other file systems

- provides the methods to create / move / copy / delete files, links and directories

- works with Java I/O and Java NIO

- gives access to security attributes

Built on Three Classes

FileSystem is an abstraction of a file system

- can close the file system or query if it is opened / read only

- provides technical information: the root directories, the used separator

- get the stores as FileStore objects

- create a path in that file system

- create a watch service

Built on Three Classes

FileStore is an abstraction of a file store within a file system

- provides the name and type of the store

- provides information on the space of the store

- gives access to security attributes

And a Factory Class

There is a 4th one: FileSystems

A factory class to create file systems

That calls methods from the FileSystemProvider class

Understanding File Systems and Providers

```
List<FileSystemProvider> providers =
    FileSystemProvider.installedProviders();

FileSystem default =
    providers.get(0).getFileSystem(URI.create("file:///"));
```

First we can check for the installed file system providers

Usually there are two: the default file system provider and the JAR or ZIP file system provider

We can then get the default (disk) file system from the right provider

```
FileSystem default1 = FileSystems.getDefault();

FileSystem default2 =
    FileSystems.getFileSystem(URI.create("file:///"));
```

We can also get the default (disk) file system with the factory class

Or by providing the right URI

```
FileSystem default = FileSystems.getDefault();

Iterable<Path> roots = default.getRootDirectories(); // C:\ D:\ E:\

Iterable<FileStore> stores = default.getFileStores();
```

We can also get the root directories from a file system

And also the file stores with more information

```
FileStore store = stores.iterator().next();

store.name(); // Data_1
store.type(); // NTFS
```

First, let us read the first file store

We can get the name of this store, and its type

Creating Files with the File Systems API

Two ways of reading / writing files:

- the Java I/O way, based on input / output streams and readers / writers

- the Java NIO way, based on channels, asynchronous or not

Fully supported by the file system API

Entry point: the FileSystemProvider class

Java I/O Operations

Two methods:

- **newInputStream(Path, OpenOption)**
- **newOutputStream(Path, OpenOption)**

Java NIO Operations

Three methods:

- newFileChannel(Path, ...)
- newByteChannel(Path, ...)
- newAsynchronousFileChannel(Path, ...)

More Methods on File System Provider

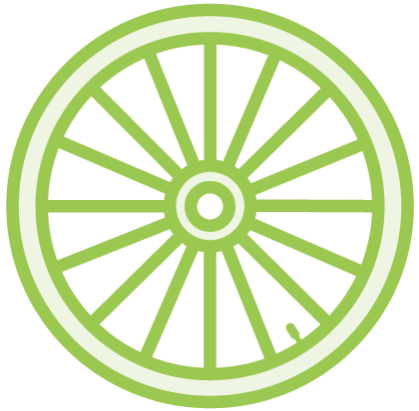
File system elements:

- files, directory
- symbolic links

Supported operations:

- create
- copy, move
- delete, delete if exists

Access to security elements



Three patterns to create directories

- fileSystem.createDirectory(file)

- fileSystemProvider.createDirectory(uri)

- fileSystemProvider.createDirectory(path)

```
File dir = new File("D:/tmp");

FileSystem defaultFS = FileSystems.getDefault();
defaultFS.createDirectory(dir);
```

If we know this directory as a file

Then we can create our directory using the file system

```
URI dir = URI.create("file:///D:/tmp");

FileSystemProvider fsp = ...;
fsp.createDirectory(dir);
```

If we know this directory as a URI

A URI “knows” which file system it belongs to

This time the directory creation is done through the file system provider

```
Path dir = Paths.get("D:/tmp");

FileSystemProvider fsp = ...;
fsp.createDirectory(dir); // ???
```

```
public static Path get(String first, String... more) {
    return FileSystems.getDefault().getPath(first, more);
}
```

If we know this directory as a path

What file system will be chosen by the API?

It depends on the way the Path has been created...

```
Path dir = Paths.get(URI.create("file:///D:/tmp"));

FileSystemProvider fsp = ...;
fsp.createDirectory(dir);
```

If we know this directory as a path

What file system will be chosen by the API?

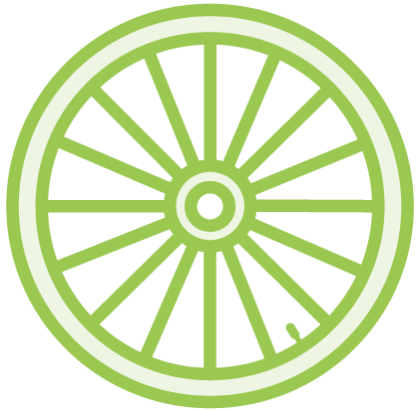
It depends on the way the Path has been created...

A Path Is Bound to a File System

Created from a String: bound to the default file system

Created from a URI: bound to the file system with the corresponding scheme

Accessing File Attributes

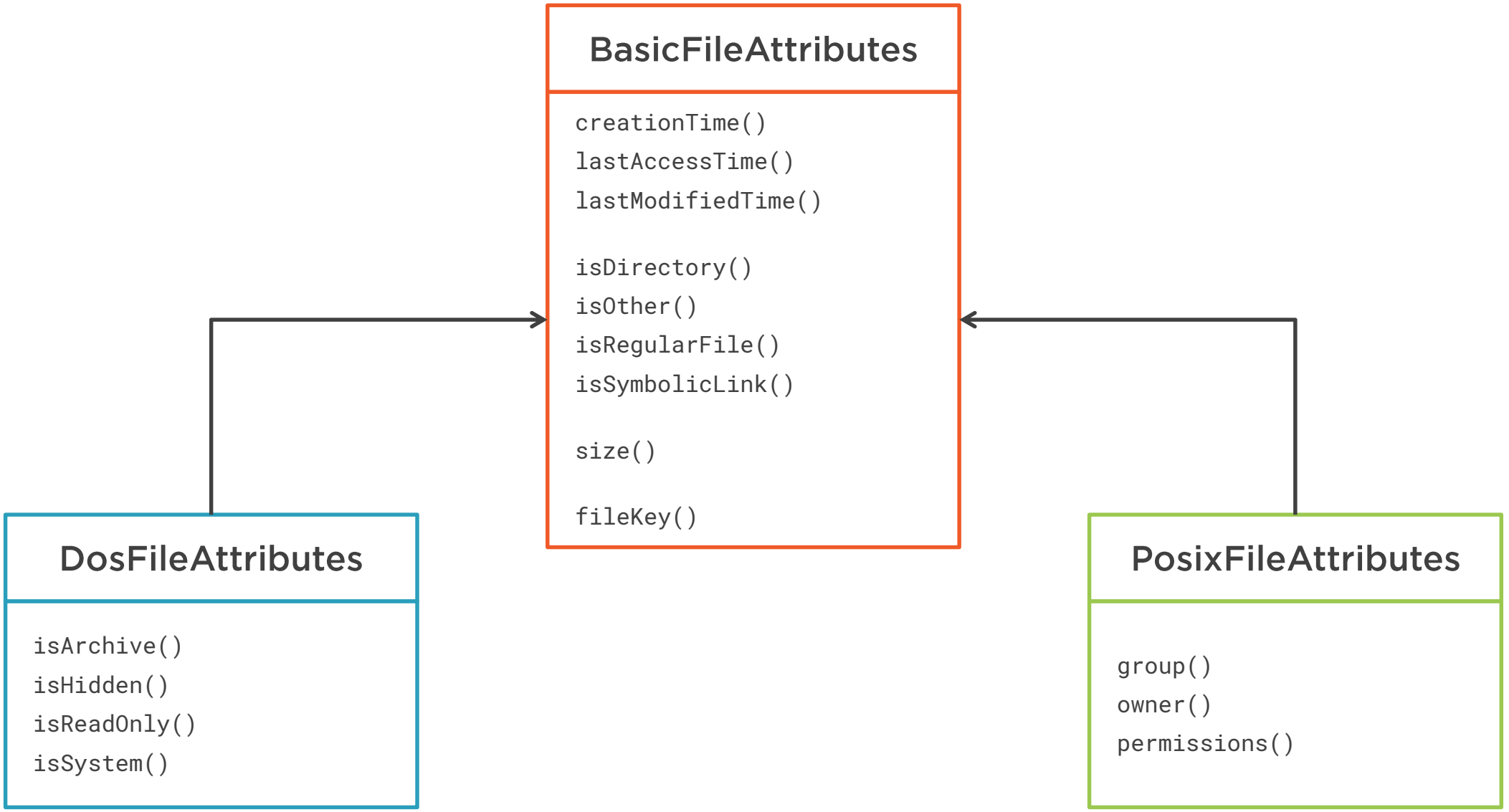


Java NIO2 gives access to file attributes both for Windows and Unix file systems

Three interfaces are involved

Reading the attributes of a file is done through the file system provider

There is a factory method in Files


BasicFileAttributes
creationTime()
lastAccessTime()
lastModifiedTime()
isDirectory()
isOther()
isRegularFile()
isSymbolicLink()
size()
fileKey()
DosFileAttributes
isArchive()
isHidden()
isReadOnly()
isSystem()
PosixFileAttributes
group()
owner()
permissions()

```
Path path = Paths.get("D:/tmp/file.txt"));
FileSystem fileSystem = path.getFileSystem();
FileSystemProvider fileSystemProvider = fileSystem.provider()
DosFileAttributes attributes =
   fileSystemProvider.readAttributes(path, DosFileAttributes.class);
```

First create a path

We can query the file system provider for the attributes of this path

We can get the file system from the path...

... and the file system provider from the file system

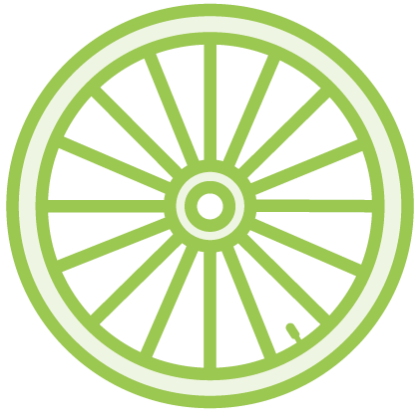
```
Path path = Paths.get("D:/tmp/file.txt"));

DosFileAttributes attributes =
   Files.readAttributes(path, DosFileAttributes.class);
```

We can also use the factory method from the Files class

Uses the right file system / file system provider to get the attributes

The JAR File System



The second file system provided by the JDK is the JAR File System

That can be used to read / write ZIP files very easily

It follows the same pattern as the disk file system

There are two modes:

- creation of ZIP files

- reading of ZIP files

A ZIP file can be written or read in two ways:

- by copying existing files in it

```
URI zipFile = URI.create("jar:file:///D:/tmp/archive.zip"));
Map<String, String> options = new HashMap<>();
options.put("create", "true");

FileSystem zipFS = FileSystems.newFileSystem(zipFile, options);
```

First create the right URI, with `jar:file:` as a scheme

Then create a map of options (`create`, `encoding`)

Then create the file system with the options

```
Path someText = Paths.get("files/some.txt");

Files.copy(someText, zipFS.getPath("some.txt"));
```

To copy a file in a ZIP archive, we first get the path of an existing file

Then we copy it to the given path within ZIP file system

Copying in a directory within the ZIP file system requires to first create that directory

```
Path dir = zipFS.getPath("files");


zipFS.provider().createDirectory(dir);
Files.copy(someText, zipFS.getPath("some.txt"));
```

Two ways to create this directory:

- using the ZIP file system

```
Path dir =
   Paths.get(URI.create("jar:file:///D:/tmp/archive.zip!/files"));

zipFS.provider().createDirectory(dir);
Files.copy(someText, zipFS.getPath("some.txt"));
```

Two ways to create this directory:

- using the ZIP file system

- providing the URI of the directory

There are two modes:

- creation of ZIP files

- reading of ZIP files

A ZIP file can be written or read in two ways:

- by copying existing files in it

- by writing content directly in it

```
Path target = zipFS.getPath("ints.bin");

OutputStream os = zipFS.provider().newOutputStream(target,
   StandardOpenOption.CREATE_NEW, StandardOpenOption.WRITE);
```

To add content to a ZIP file, we first create an entry in this file system

Then we create an output stream on this entry, from the file system provider

This output stream can be used to create any other stream as usual

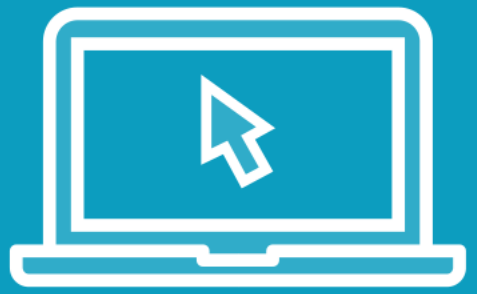
```
Path target = zipFS.getPath("ints.bin");

ByteChannel channel = zipFS.provider().newByteChannel(target, options);
```

The pattern is almost the same for the creation of a byte channel

In this call, options is a set instead of a varargs, that has to be created

Demo



Let us see some code!

Let us see those file systems in action

First the classical disk file system

Then the ZIP file system

Module Wrap Up

What did you learn?

The NIO2 File System API

How to handle file systems with native elements in the JDK

How to manipulate files using this API

How to manipulate ZIP files