Java Fundamentals: Input/Output

INTRODUCTION TO JAVA I/O, ORGANIZATION OF THE API

José Paumard

PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard https://github.com/JosePaumard

Accessing Files & Network

Java I/O

Java 1 (1996)

Java NIO

Java 4 (2002)

Java NIO 2

Java 7 (2011)

Accessing Files & Network

Java I/O

Java 1 (1996)

Java NIO

Java 4 (2002)

Java NIO 2

Java 7 (2011)

Accessing Files & Network

Java I/O

Java 1 (1996)

Java Fundamentals: NIO & NIO2

Java NIO

Java 4 (2002)

Java NIO 2

Java 7 (2011)

Java I/O: how to read and write information

- bytes and chars
- on many media: disk, network, memory

The API is a little complex

But very well structured

Built on the decorator pattern

This is a Java course

Fair knowledge of the language and its main API

The Collection framework

This is a fundamental course

Agenda

Introducing the big picture: the structure of the API, files and paths

Reading chars, the decorator pattern, exceptions

Writing chars, flushing and closing, using formats

Reading and writing bytes & primitive types

Reading and writing objects, serialization

Sending data on the network, hybrid streams

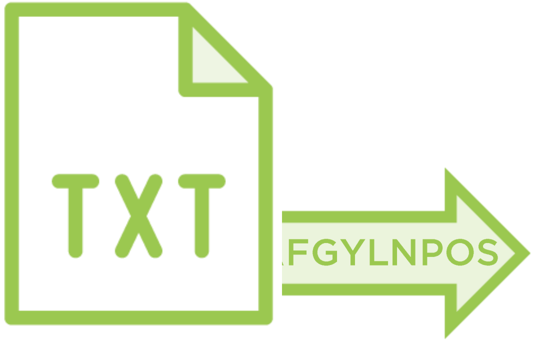
Introducing Java I/O

Java I/O is organized on four base classes

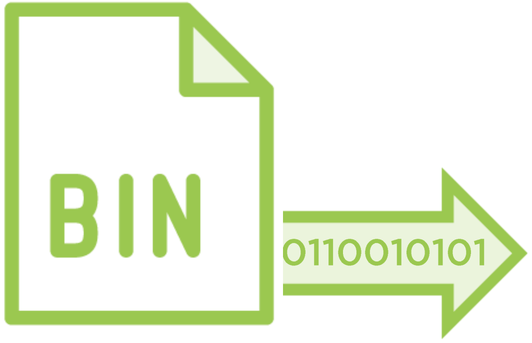
- Reader and Writer
- InputStream and OutputStream

And two utility classes:

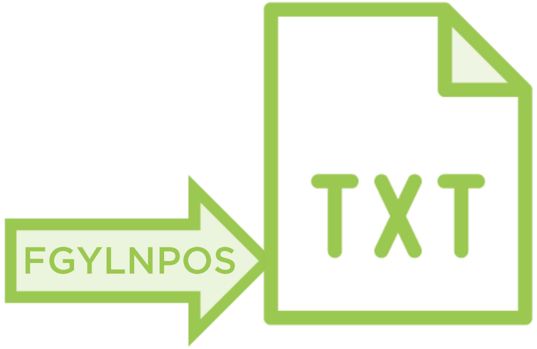
- File
- Path



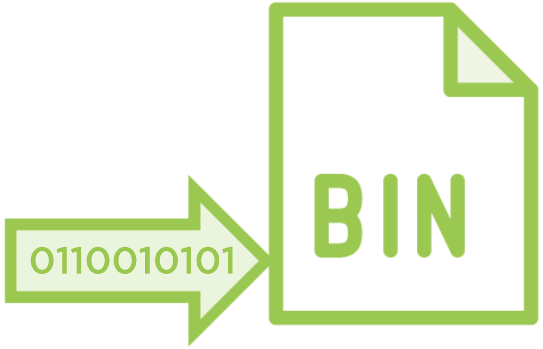
Reader



InputStream



Writer



OutputStream

The File Object

File and Path are models of a file or directory on the disk

File is the first class introduced

```java
File file = new File("files/data.txt");
```

The File class is a file

1) creating a File object does not create anything on the disk

2) a File object can be a file or a directory

```
File file = new File("files/data.txt");
file.exists();
file.isDirectory();
file.isFile();

file.canRead();
file.canWrite();
file.canExecute();
```

There is a set of more than 30 methods to test for this file

To test the existence or the nature of this file

```java
File file = new File("files/data.txt");
file.createNewFile();
file.mkdir();
file.mkdirs();

file.delete();
file.deleteOnExit();
file.renameTo("files/file.txt");
```

To create, touch or modify this file

```
File file = new File("files/data.txt");

file.getName();
```

The name of a file is its name without the path

```java
File file = new File("files/data.txt");

file.getName();

file.getParent();
```

The parent of a file is the path before its name

It can be relative: `files/data.txt` **returns** `files`

Or absolute: `/tmp/files/data.txt` **returns** `/tmp/files`

```java
File file = new File("files/data.txt");

file.getName();

file.getParent();

file.getPath();
```

The path of a File is the String representing this file

```
File file = new File("files/data.txt");

file.getAbsolutePath();
```

The absolute path of a File is this path if the File is absolute, and resolved in an OS dependent way if not

The parent path can be the user directory or the current directory

```java
File file = new File("files/data.txt");

file.getAbsolutePath();

file.getCanonicalPath();
```

The canonical path is unique for a given file, computed in an OS dependent way

The use of . and .. directories is resolved and simplified

The use of symbolic links is also resolved

The Path Object

The Path interface has the same kind of methods as the File class

It is a Java 7 interface

- to check for the file / directory

- to create / touch / delete it

- to get its name / path / etc…

- can get the attributes of the file

- and other methods to check for directory events

Other interesting methods

- normalize(): removes redundant elements
- toAbsolutePath()
- toRealPath(): resolves the symbolic links
- Files.isSame(path1, path2)
- resolve() and relativize()

Relativizing two paths consists in creating a relative path from one to the other

```
Path root  = Paths.get("files");
Path child = Paths.get("data.txt");

Path resolved = root.resolve(child); // files/data.txt
```

Resolving: given two relative paths

Resolving the path concatenates them

```
Path root  = Paths.get("files");
Path child = Paths.get("D:/tmp/data.txt");

Path resolved = root.resolve(child); // D:/tmp/data.txt
```

If the child path is absolute

Then the resolved path is the child path

```java
Path file  = Paths.get("files/data1.txt");
Path sibling = Paths.get("data2.txt");

Path resolved = file.resolveSibling(sibling); // files/data2.txt
```

One can also resolve siblings

Then the resolved path sees the other file as a sibling of the first one

```
Path dir  = Paths.get("D:/src/java");
Path file = Paths.get("D:/src/java/org/paumard/Main.java");

Path relative = dir.relativize(file); // org/paumard/Main.java
```

Relativizing is about finding a path from a source to a target

Then the result is the relative path of the child in the directory

```
Path dir  = Paths.get("src/java");
Path file = Paths.get("src/java/org/paumard/Main.java");

Path relative = dir.relativize(file); // org/paumard/Main.java
```

In the case where the paths are not absolute

Then the result is the same as previously if the parent is the same

```
Path dir  = Paths.get("src/java");
Path file = Paths.get("project/src/java/org/paumard/Main.java");

Path relative = dir.relativize(file);
// ..\..\project\src\java\org\paumard\Main.java
```

In the case where the paths are not absolute

It also work if there are no common elements between the two paths

Summary of the relativize method:

- both paths can be absolute

- both paths can be relative

- if one path is absolute and not the other, then an IllegalArgumentException is raised

Module Wrapup

What did we learn?

The Java I/O organization

The notion of file, with the File class and the Path interface

Some interesting methods of the Path interface