Visiting Directory Trees

José Paumard

PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard https://github.com/JosePaumard

Agenda

Presentation of the API from NIO2 to explore the content of directories

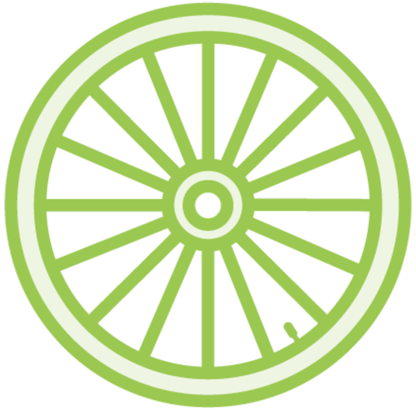
In an efficient way

Path matchers

How to filter the content of a tree

How to visit a directory tree

Directory Streams and Matchers



A directory stream is a way of analyzing the content of a directory

Without exploring the subdirectories

It can be used to get all the content

And can also filter its content

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


DirectoryStream<Path> directoryStream =
    Files.newDirectoryStream(dir, path -> Files.isDirectory(path));
```

First, create the path to the right directory

Then call the factory method from the Files class

It takes a lambda that is a filter object (not the java.util one)

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

DirectoryStream<Path> directoryStream =
    Files.newDirectoryStream(dir, Files::isDirectory);
```

First create the path to the right directory

Then call the factory method from the Files class

It takes a lambda that is a filter object (not the java.util one)

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


DirectoryStream<Path> directoryStream =
    Files.newDirectoryStream(dir, "*.java");
```

We can also pass a regular expression to match the file & directory names

```
Path dir = Paths.get("D:/files");
PathMatcher pathMatcher =
    FileSystems.getDefault().getPathMatcher("glob:**/*.java");
DirectoryStream<Path> directoryStream =
    Files.newDirectoryStream(dir, pathMatcher::matches);
```

If we need complex file name checking, we can use a file matcher

It has a matches method, that takes a path and returns a boolean

The path matcher allows for two kinds of regular expression:

- `regex:` specified in the Pattern class

- `glob:` which is a simplified version of `regex:`, specified in the FileSystem.getPathMatcher method

There might be more in the future

```
DirectoryStream<Path> directoryStream = ...;

for (Path path: directoryStream) {
   // operations on the elements
}
```

What can be done with this directory stream?

It is an iterable, so we can use the for each syntax

```java
DirectoryStream<Path> directoryStream = ...;

directoryStream.forEach(System.out::println);
```

What can be done with this directory stream?

It is an iterable, so we can use the for each syntax

We can also call its forEach() method directly

```java
DirectoryStream<Path> directoryStream = ...;

List<Path> paths =
StreamSupport.stream(directoryStream.spliterator(), false)
             .collect(Collectors.toList());
```

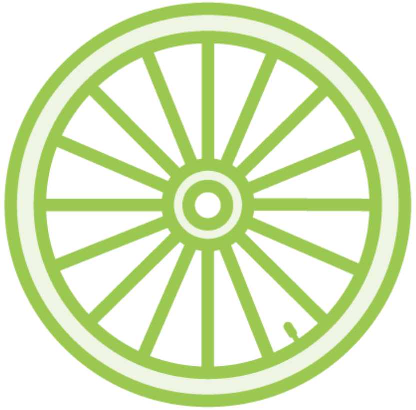
What can be done with this directory stream?

It is an iterable, so we can use the for each syntax

We can also call its forEach() method directly

Or create a regular Stream and process it

Walking Directory Trees



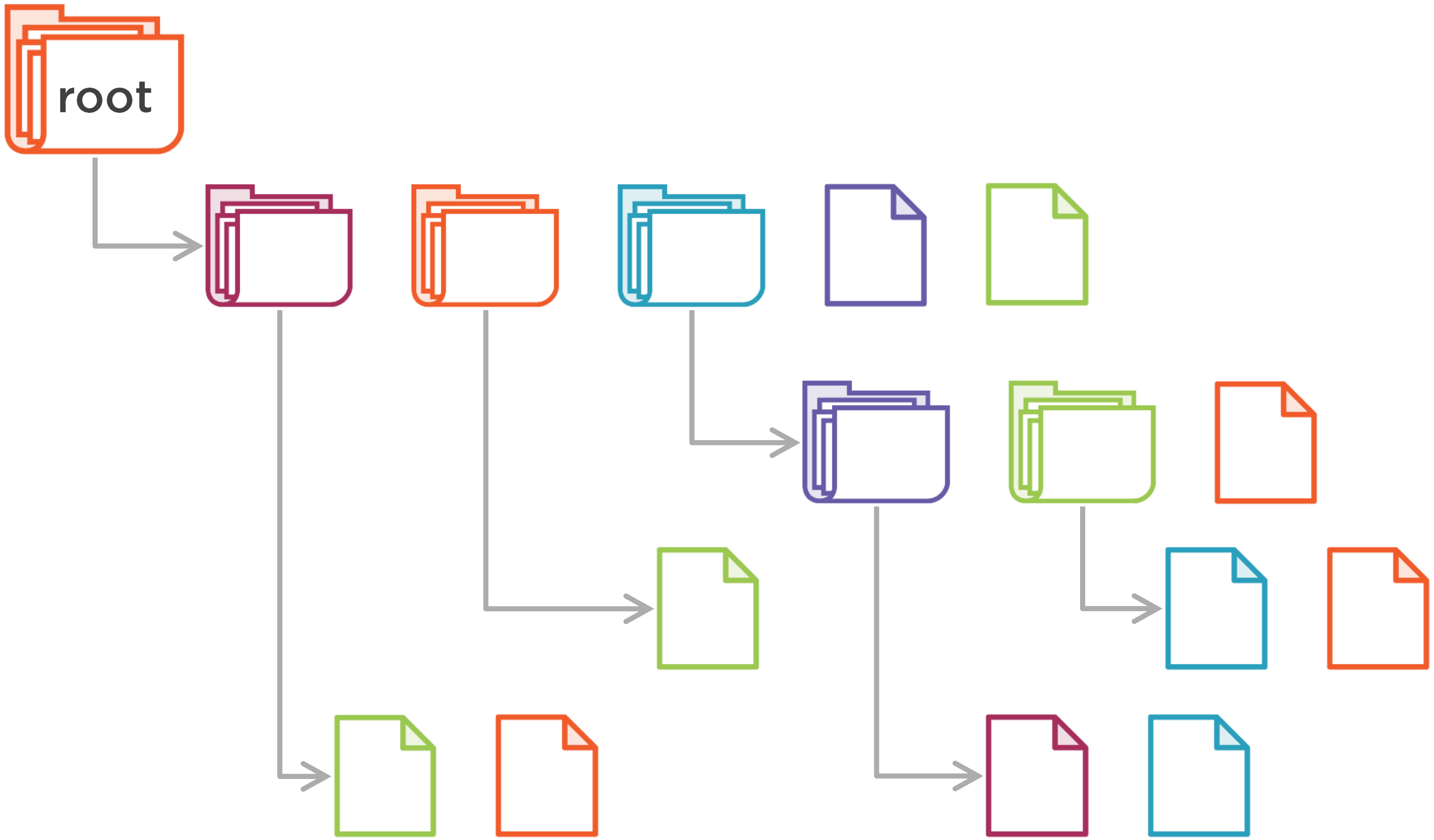
Walking a directory tree consists in exploring all the files and subdirectories

It can be done in two ways:

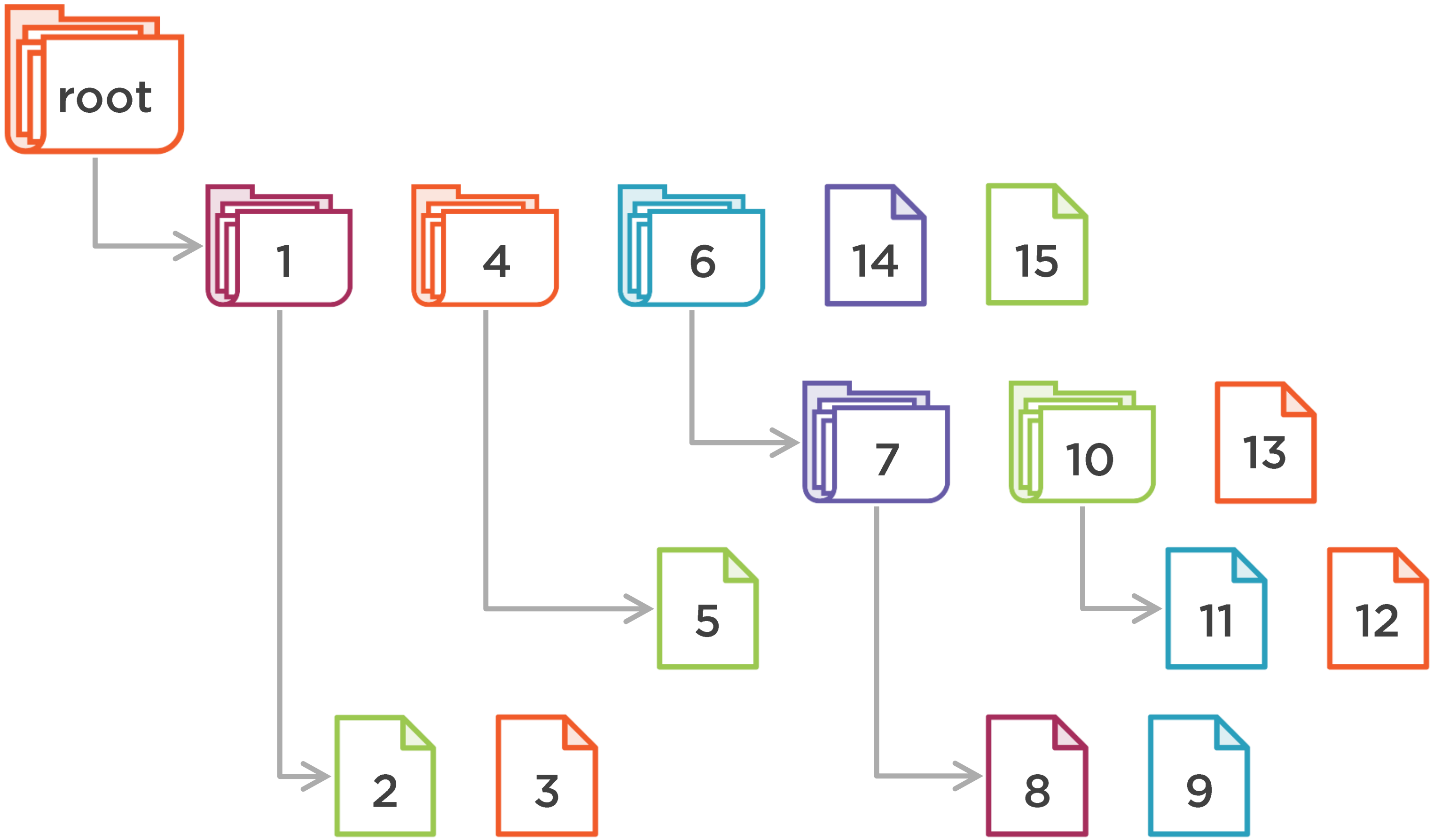
- using a depth-first approach

- or a breadth-first approach

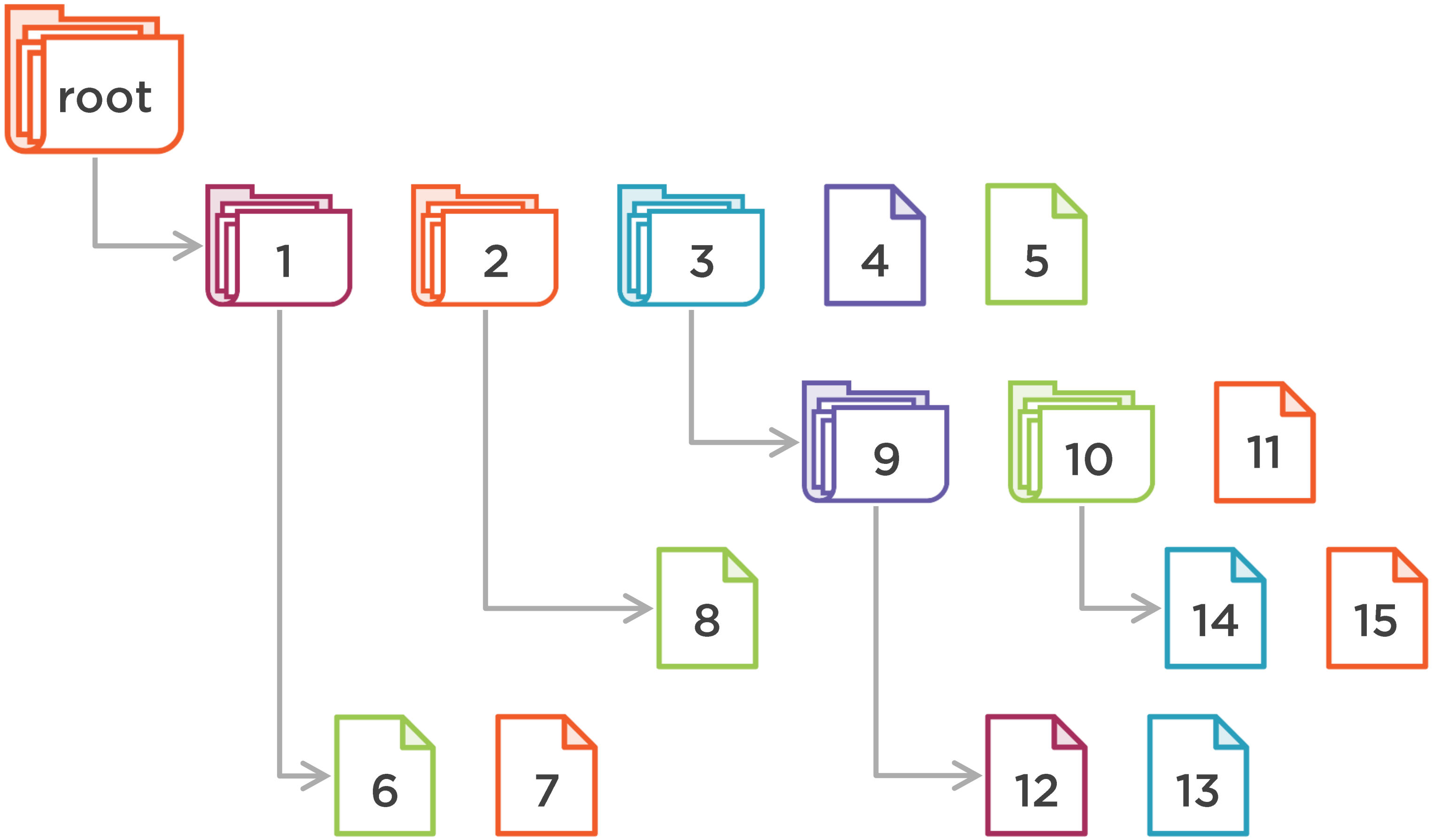
Java does it using the depth-first approach


root

Depth First



Breadth First



The Files.walk methods walk a directory tree

The parameters are:

- the starting point as a path

- the maximum depth to be explored

- an option to follow the links or not

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

Stream<Path> paths =
    Files.walk(dir);
```

The basic pattern just takes the starting path

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

Stream<Path> paths =
    Files.walk(dir, 3);
```

The basic pattern just takes the starting path

Then it can take a maximum depth

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

Stream<Path> paths =
    Files.walk(dir, 3, FileVisitOption.FOLLOW_LINKS);
```

The basic pattern just takes the starting path

Then it can take a maximum depth

And we can specify whether or not to follow the symbolic links

In case there are cycles an exception is raised

The Files.find method works the same

It takes a BiPredicate as a parameter

And returns a stream of the matching paths

```java
Path dir = Paths.get("D:/sources");
PathMatcher pathMatcher =
    FileSystems.getDefault().getPathMatcher("glob:**/*.java");

Stream<Path> paths =
    Files.find(dir,
               (path, attributes) -> pathMatcher.matches(path));
```

The basic find pattern just takes

- the starting path
- the bipredicate (from java.util)
- whether or not to follow the symbolic links

```java
Path dir = Paths.get("D:/sources");
PathMatcher pathMatcher =
    FileSystems.getDefault().getPathMatcher("glob:**/*.java");

Stream<Path> paths =
    Files.find(dir,
               (path, attributes) -> pathMatcher.matches(path));
```

The nice thing is that we can find files by their attributes

For instance all the files modified since a given date

Or all the files belonging to a certain user

The streams are lazily built while walking through the directory trees

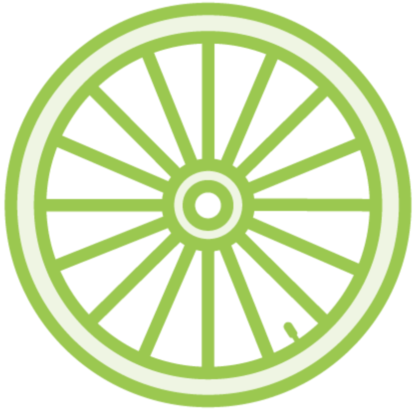
Meaning they are weakly consistent

The file system might change during the process

Even delete a directory currently explored

Thus raising an exception

Visiting Directory Trees



Visiting is different from Walking

Visiting a directory tree offers more control over the process:

- it can be interrupted

- it can skip elements based on filtering

The pattern uses the Files.walkFileTree methods

The parameters are:

- a file visitor

- whether or not to follow the links

- a maximum depth of exploration

The File Visitor

A file visitor is used during the traversal of a tree

- it can act when a directory is met, before and after it has been visited

- it tells what to do with every file

- it can handle exceptions

The File Visitor

There is an interface: FileVisitor

And an adapter class

With four methods to implement

```java
Path dir = Paths.get("D:/sources");
FileVisitor<Path> fileVisitor = ...;

Files.walkFileTree(dir, fileVisitor);
```

As usual we need a starting directory

And a file visitor

Calling walkFileTree returns the starting directory

```
public interface FileVisitor<T> {

    FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs);

    FileVisitResult postVisitDirectory(T dir, IOException exc);

    FileVisitResult visitFile(T file, BasicFileAttributes attrs);

    FileVisitResult visitFileFailed(T file, IOException exc);

}
```

The FileVisitor interface has four simple methods

Two for handling directories

And another two for handling files

Note: all these methods throw IOException

```
public enum FileVisitResult {

   CONTINUE,

   TERMINATE,

   SKIP_SUBTREE,

   SKIP_SIBLINGS

}
```

CONTINUE: to continue to visit the directory tree

TERMINATE: to end up the process now

SKIP_SUBTREE: to prevent the exploration of the current directory

SKIP_SIBLINGS: to stop exploring the current directory

```
public class FileFinder implements FileVisitor<Path> {
    private String searchedFileName;

    FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) {
        return CONTINUE;
    }
}
```

Let us create a file visitor to find a given file and stop when it is found

We need to visit all the directories

```java
public class FileFinder implements FileVisitor<Path> {
    private String searchedFileName;

    FileVisitResult postVisitDirectory(Path dir, BasicFileAttributes attrs) {
        return CONTINUE;
    }
}
```

Let us create a file visitor to find a given file and stop when it is found

We need to visit all the directories

```
public class FileFinder implements FileVisitor<Path> {
    private String searchedFileName;

    FileVisitResult visitFileFailed(Path path, IOException exc) {
        return CONTINUE;
    }
}
```

If we cannot visit a file we can just continue

```
FileVisitResult visitFile(Path path, BasicFileAttributes attrs) {
    if (path.getName().equals(searchedFileName)) {
        this.found = path;
        return TERMINATE;
    } else {
        return CONTINUE;
    }
}
```

And then when we found the searched file, we just stop

We can add a getter of the found field to access it once the walkFileTree() method returns

Demo

Let us see some code!

Let us play with this directory walking API

And set up a system to visit a directory tree and extract information from it

Module Wrap Up

What did you learn?

How to explore directory trees

To conduct this exploration in a lazy way

Without blocking the whole file system

Using different patterns for different needs