Notatki - NIO.2 Files

Spis treści

NIO.2 i Files

No dobrze, umiemy się już poruszać po ścieżkach. Teraz możemy przejść do omówienia dostępnych klas "po nowemu" w kontekście faktycznego dostępu do plików i operowania na nich.

W tym celu wprowadzimy sobie klasę `java.nio.file.Files`. Tutaj zwracam uwagę na to podobieństwo między `Path` i `Paths`, a także `File` i `Files`. `Paths` i `Files` są klasami pomocniczymi, które będziemy wykorzystywać aby mieć dostęp do ich metod statycznych. `Path` i `File` są klasami, które reprezentują ścieżkę oraz plik na dysku.

Klasa `Files` dostarcza nam metody, które już poznaliśmy. Metody te pozwalają nam wykonać operacje takie jak przeniesienie pliku między katalogami, albo zmiana jego nazwy. Dlatego też najpierw poruszymy te metody, a potem zestawimy je sobie w tabelce z tymi, które już poznaliśmy w klasie `File`.

exists()

Najprostsza metoda, która pozwala sprawdzić, czy podana ścieżka faktycznie może zostać znaleziona na dysku. Możemy za jej pomocą sprawdzić występowanie zarówno pliku jak i katalogu. Metoda zwraca `boolean`.

```
Path path = Paths.get("zajavka/myFile.txt");
System.out.println(Files.exists(path)); ①
```

① Wydrukowane zostanie: `true` albo `false`.

createDirectory()

Jak sama nazwa wskazuje, metoda ta służy do stworzenia katalogu. Jeżeli chcemy stworzyć po drodze kilka katalogów, bo przykładowo połowa ścieżki nie istnieje - do tego mamy metodę poniżej, czyli `createDirectories()`. Typem zwracanym z metody jest `Path`.

```
Path createDirectory = Paths.get("zajavka/newDir");
try {
    Files.createDirectory(createDirectory); ①
} catch (IOException e) {
    System.err.println("Files.createDirectory(createDirectory): " + e.getMessage());
}
```

① Wynikiem działania metody będzie stworzenie katalogu `newDir` pod warunkiem, że katalog `zajavka` istniał już wcześniej.

createDirectories()

Jeżeli po drodze występuje kilka katalogów, które nie istnieją, a my chcemy stworzyć wszystkie po drodze do tego ostatniego, który nas interesuje to poprzednia metoda nie odniesie efektu (zostanie wyrzucony wyjątek). Należy wtedy wykorzystać `createDirectories()`. Typem zwracanym z metody jest `Path`.

```
Path createDirectories = Paths.get("zajavka/newDir/newDir2/newDir3");
try {
    Files.createDirectories(createDirectories); ①
} catch (IOException e) {
    System.err.println("Files.createDirectories(createDirectory): " + e.getMessage());
}
```

① Wynikiem działania metody będzie stworzenie katalogu katalogów wymienionych na ścieżce przypisanej do zmiennej `createDirectories`.

copy()

Ta metoda jak sama nazwa wskazuje służy do kopiowania ☺. Typem zwracanym z metody jest `Path`. Przykład kopiowania pliku.

```
Path oldPath = Paths.get("zajavka/myFile.txt");
Path newPath = Paths.get("zajavka/myNewFile.txt");
try {
    Files.copy(oldPath, newPath); ①
} catch (IOException e) {
    System.err.println("Files.copy(oldPath, newPath): " + e.getMessage());
}
```

① Wynikiem działania metody będzie stworzenie kopii pliku określonej pod ścieżką `newPath`.

A co z kopiowaniem katalogów? Jeżeli chcemy kopiować katalogi, to nastąpi tak zwane "shallow copy", czyli zawartość katalogu nie zostanie skopiowana, a tylko sam katalog.

W takim przypadku musimy sami zatroszczyć się o kopiowanie odpowiednich plików. Domyślnie też, jeżeli plik, który kopiujemy istnieje już w miejscu docelowym na dysku, nie zostanie on nadpisany.

Trzeba to wymusić przez podanie odpowiednich parametrów do metody. Nie chcę natomiast schodzić na tyle głęboko w temat, bo jak będzie to potrzebne w pracy to i tak będziesz to Googlować ☺.

move()

Jak sama nazwa wskazuje, metoda ta służy do przeprowadzki pliku. Generalnie to jest o tyle sprytne, że za pomocą tej metody można również zmienić nazwę pliku, bo przecież filozoficznie rzecz biorąc, możemy przenieść plik do tego samego katalogu, ale pod inną nazwą pliku i też będzie to move ☺.

Tak samo jak w poprzednim przykładzie jeżeli plik, który przenosimy istnieje już w miejscu docelowym na dysku, nie zostanie on nadpisany. Trzeba to wymusić przez podanie odpowiednich parametrów do metody. Nie chcę natomiast schodzić na tyle głęboko w temat, bo jak będzie to potrzebne w pracy to i tak będziesz to googlować ☺. Typem zwracanym z metody jest `Path`.

```
Path oldPath = Paths.get("zajavka/myFile.txt");
Path newPath = Paths.get("zajavka/myDir/myNewFile.txt");
try {
    Files.move(oldPath, newPath);
} catch (IOException e) {
    System.err.println("Files.move(oldPath, newPath): " + e.getMessage());
}

Path oldPath1 = Paths.get("zajavka/myDir");
Path newPath1 = Paths.get("zajavka/myNewDir");
try {
    Files.move(oldPath1, newPath1);
} catch (IOException e) {
    System.err.println("Files.move(oldPath1, newPath1): " + e.getMessage());
}
```

Wynikiem działania powyższych przykładów będzie przeniesienie plików/katalogów ze starych ścieżek na nowe ścieżki.

delete i deleteIfExists()

Raczej nie będzie to zaskoczenie, jak napiszę, że ta metoda usuwa zasób znajdujący się pod podaną ścieżką. Ale... trzeba pamiętać, że występuje pare przypadków, w których takie usunięcie nie może zostać wykonane.

Żeby nie uczyć się wszystkich na pamięć, najważniejsze jest zapamiętanie, że jeżeli ścieżka reprezentuje katalog, w którym znajdują się pliki - wywołanie metody wyrzuci wyjątek typu `IOException`.

Do tego przykładowo, gdy ścieżka, którą chcemy usunąć nie istnieje - również zostanie wyrzucony wyjątek, chyba, że użyjemy metody `deleteIfExists()`. Metoda `deleteIfExists()` nie wyrzuca wyjątku, gdy zasób który chcemy usunąć nie istnieje. Zamiast tego zwróci `boolean` sugerujący czy usunięcie się powiodło czy nie.

```
Path path1 = Paths.get("zajavka/myFile.txt");
Path path2 = Paths.get("zajavka/myDir");

try {
    Files.delete(path1);
} catch (IOException e) {
    System.err.println("Files.delete(path1): " + e.getMessage());
}

try {
    Files.delete(path2);
} catch (IOException e) {
    System.err.println("Files.delete(path2): " + e.getMessage());
}
```

Metoda `delete()` nic nie zwraca, a efektem jej działania jest usunięcie podanej ścieżki, pod warunkiem, że ścieżka istnieje.

A jeżeli mamy przypadek próby usunięcia zasobu, którego faktycznie nie ma:

```
Path path3 = Paths.get("zajavka/nonExisting.txt");

try {
    Files.deleteIfExists(path3);
} catch (IOException e) {
    System.err.println("Files.deleteIfExists(path3): " + e.getMessage());
}
```

readAllLines()

Tak jak mówi nazwa metody, dzięki niej możemy odczytać wszystkie linijki z pliku. Natomiast trzeba z tą metodą uważać, bo wszystkie linijki z pliku są odczytywane na raz i zapisywane do listy Stringów.

Jeżeli plik jest wystarczająco duży, możemy dostać `OutOfMemoryError`. Dalej będzie pokazana metoda, która pozwala poradzić sobie z takimi przypadkami. Także `readAllLines()` stosujmy raczej do małych plików.

```
Path path = Paths.get("zajavka/myFile.txt");
try {
    List<String> allLines = Files.readAllLines(path); ①
    allLines.forEach(line -> System.out.println("Line: " + line)); ②
} catch (IOException e) {
    System.err.println("Files.readAllLines(path): " + e.getMessage());
}
```

① Wczytanie całej zawartości pliku jako lista Stringów.

② Wydrukowanie każdej linijki z listy na ekranie.

isSameFile()

Jeżeli chcemy sprawdzić czy 2 ścieżki wskazuję na ten sam zasób w naszym systemie plików, używamy do tego `isSameFile()`. Mówiąc zasób, mam też na myśli katalogi.

W praktyce metoda ta działa tak, że w pierwszej kolejności sprawdza, czy ścieżki są sobie równe w rozumieniu metody `equals()` (nie następuje tutaj żadna weryfikacja czy zasoby istnieją fizycznie na dysku). Jeżeli takie sprawdzenie zwróci `false`, następuje fizyczne sprawdzenie, czy ścieżki na dysku wskazują na ten zasób. Należy tutaj dodać, że metoda `isSameFile()` nie sprawdzana zawartości plików, służy do sprawdzania ścieżek. Jeżeli któryś z plików nie istnieje na dysku, zostaje wyrzucone `IOException`.

Przykład kodu:

```
Path path1 = Paths.get("zajavka/someDirectory/isSameFile.txt");
Path path2 = Paths.get("zajavka/someDirectory/isSameFile.txt");

Path path3 = Paths.get("zajavka/someDirectory/testingDir/../isSameFile2.txt");
Path path4 = Paths.get("zajavka/someDirectory/isSameFile2.txt");

Path path5 = Paths.get("zajavka/./someDirectory/./isSameFile2.txt");
Path path6 = Paths.get("zajavka/someDirectory/isSameFile2.txt");

Path path7 = Paths.get("zajavka/someDirectory/someDirectory1");
Path path8 = Paths.get("zajavka/someDirectory/someDirectory2");

try { ①
    System.out.println(Files.isSameFile(path1, path2));
} catch (IOException e) {
    System.err.println("Files.isSameFile(path1, path2): " + e);
}

try { ②
    System.out.println(Files.isSameFile(path3, path4));
} catch (IOException e) {
    System.err.println("Files.isSameFile(path3, path4): " + e);
}
```

```
try { ③
    System.out.println(Files.isSameFile(path5, path6));
} catch (IOException e) {
    System.err.println("Files.isSameFile(path5, path6): " + e);
}

try { ④
    System.out.println(Files.isSameFile(path7, path8));
} catch (IOException e) {
    System.err.println("Files.isSameFile(path7, path8): " + e);
}
```

① Ścieżki `path1` oraz `path2` nie muszą istnieć na dysku. Na ekranie zostanie wydrukowane: `true`

② Jeżeli plik `isSameFile2.txt` istnieje na dysku - zostanie wydrukowane `true`. Jeżeli nie istnieje - zostanie wyrzucony wyjątek.

③ Jeżeli plik `isSameFile2.txt` istnieje na dysku - zostanie wydrukowane `true`. Jeżeli nie istnieje - zostanie wyrzucony wyjątek.

④ Jeżeli ścieżki `path7` i `path8` istnieją na dysku - zostanie wydrukowane `false`. Jeżeli nie istnieją - zostanie wyrzucony wyjątek.

New Buffered readers

Kolejne Streamy i znowu pojawia się pytanie po co. Bo z każdą nową wersją, którą wprowadzają twórcy, ich założeniem jest poprawić wydajność rozwiązania i często możliwie skrócić zapis kodu dla danego mechanizmu. Tak też jest w tym przypadku.

```
Path readingPath = Paths.get("/zajavka/bufferedReadingFile.txt");
try (BufferedReader reader = Files.newBufferedReader(readingPath)) {
    String line = null;
    while((line = reader.readLine()) != null) {
        System.out.println("nextLine: " + line); ①
    }
} catch (IOException e) {
    e.printStackTrace();
}

Path writingPath = Paths.get("/zajavka/bufferWritingFile.txt");
List<String> data = new ArrayList<>();
try (BufferedWriter writer = Files.newBufferedWriter(writingPath)) {
    writer.write("zajavka!"); ②
} catch (IOException e) {
    e.printStackTrace();
}
```

① Na ekranie zostanie wydrukowana cała zawartość pliku pod ścieżką `readingPath`.

② Do pliku pod ścieżką `writingPath` zostanie zapisana treść `zajavka!`.

File Attributes

W tej sekcji chciałem poruszyć zagadnienie funkcjonalności, które dostarcza nam klasa `Files` w odniesieniu do atrybutów plików. Wcześniej cały czas pokazywaliśmy sobie metody dotyczące ścieżek, teraz porozmawiamy o atrybutach. Inaczej mówiąc, porozmawiamy o metadanych (czyli danych o danych) w odniesieniu do plików i/lub katalogów. Czyli o informacjach takich jak przykładowo data utworzenia pliku, nie będziemy tutaj poruszać kwestii zawartości pliku. Tutaj należy też dodać, że niektóre systemy operacyjne przechowują informacje o plikach charakterystyczne dla danego systemu operacyjnego. Czyli przykładowo mogą wystąpić atrybuty pliku, które występują w systemie Ubuntu, a nie będzie ich na Windowsie.

size()

Nazwa jest chyba wyczerpująca ☺. W ten sposób sprawdzamy rozmiar naszego pliku w bajtach. Metoda ta służy do sprawdzania rozmiaru pliku, jeżeli chcemy sprawdzić rozmiar folderu, jest to zależne od tego na jakim systemie operacyjnym pracujemy. Jeżeli nie zadziała nam sprawdzenie rozmiaru folderu przy wykorzystaniu tej metody należy dodać ręcznie do siebie rozmiar wszystkich plików w folderze.

```
try {
    System.out.println(Files.size(Paths.get("zajavka/myFile.txt"))); ①
} catch (IOException e) {
    e.printStackTrace();
}
```

① Na ekranie zostanie wydrukowany rozmiar pliku w bajtach.

getLastModifiedTime() i setLastModifiedTime()

Tutaj należy zwrócić uwagę na to, że metoda `getLastModifiedTime()` zwraca obiekt klasy `FileTime`. Później możemy wyciągnąć z tego informacje za pomocą metod `toString()`, `toMilis()` albo `toInstant()`.

Mamy również możliwość ręcznego ustawienia daty ostatniej modyfikacji pliku poprzez `setLastModifiedTime()`, która przyjmuje obiekt klasy `FileTime`. W przykładzie ustawiliśmy ostatnią datę modyfikacji w godzinach od `epoch`. Oczywiście obie te metody wyrzucą `IOException` jeżeli plik nie zostanie znaleziony.

```
Path path = Paths.get("zajavka/myFile.txt");
try {
    FileTime lastModifiedTime = Files.getLastModifiedTime(path);
    System.out.println(lastModifiedTime);

    Path path1 = Files.setLastModifiedTime(path, FileTime.from(18284L, TimeUnit.HOURS));
    FileTime lastModifiedTime2 = Files.getLastModifiedTime(path1);
    System.out.println(lastModifiedTime2); ①
} catch (IOException e) {
    e.printStackTrace();
}
```

① Na ekranie zostanie wydrukowane: `1972-02-01T20:00:00Z`

isDirectory()

Metoda jak sama nazwa wskazuje, czy podana ścieżka odnosi się do katalogu.

```
Path path1 = Paths.get("zajavka/myFile.txt");
Path path2 = Paths.get("zajavka");

boolean directory1 = Files.isDirectory(path1);
System.out.println("directory1: " + directory1); ①
boolean directory2 = Files.isDirectory(path2);
System.out.println("directory2: " + directory2); ②
```

① Na ekranie zostanie wydrukowane: `directory1: false`

② Na ekranie zostanie wydrukowane: `directory2: true`

isRegularFile()

Ta metoda z kolei wskazuje, czy podana ścieżka odnosi się do pliku.

```
Path path1 = Paths.get("zajavka/myFile.txt");
Path path2 = Paths.get("zajavka");

boolean regularFile1 = Files.isRegularFile(path1);
System.out.println("regularFile1: " + regularFile1); ①
boolean regularFile2 = Files.isRegularFile(path2);
System.out.println("regularFile2: " + regularFile2); ②
```

① Na ekranie zostanie wydrukowane: `regularFile1: true`

② Na ekranie zostanie wydrukowane: `regularFile2: false`

isHidden()

Ta metoda pozwala sprawdzić, czy plik jest ukryty. Dlatego, że możemy ukrywać pliki na dysku ☺.

```
Path path1 = Paths.get("zajavka/myFile.txt");

try {
    boolean hidden1 = Files.isHidden(path1);
    System.out.println("hidden1: " + hidden1); ①
} catch (IOException e) {
    e.printStackTrace();
}
```

① Na ekranie zostanie wydrukowane: `hidden1: false`

isReadable() i isExecutable()

W niektórych systemach operacyjnych użytkownik może nie mieć uprawnień aby odczytać plik, a tym bardziej aby go uruchomić.

W przypadku `isExecutable()` nie jest natomiast sprawdzane czy rozszerzenie pliku jest 'uruchamialne'. Czyli, przykładowo rozszerzenie `.png` nie uruchamia nam żadnego programu ale `.exe` już tak. Dlatego w poniższym przypadku mamy dwukrotnie `true`.

```
Path path1 = Paths.get("zajavka/myFile.txt");
boolean readable1 = Files.isReadable(path1);
System.out.println("readable1: " + readable1); ①

boolean executable1 = Files.isExecutable(path1);
System.out.println("executable1: " + executable1); ②
```

① Na ekranie zostanie wydrukowane: `readable1: true`

② Na ekranie zostanie wydrukowane: `executable1: true`

Przykładowe metody wprowadzone w Java 8

list()

Metoda, która służy do listowania zawartości podanej ścieżki. Przeszukiwane jest tylko jedno zagłębienie katalogu, to znaczy, że jeżeli natrafimy na inny katalog, to jego zawartość nie zostanie uwzględniona. Zwracany jest `Stream<Path>`, dlatego wspomniałem o Javie 8. Mając taki `Stream`, możemy wykonywać operacje takie jak `map()`, `filter()` itp. Warto też zauważyć, że metoda ta wyrzuca `IOException`, który musimy obsłużyć.

```
Path path = Paths.get("zajavka");
try {
    Stream<Path> list = Files.list(path);
    List<Path> absolutes = list
        .filter(p -> Files.isRegularFile(p))
        .map(p -> p.toAbsolutePath())
        .collect(Collectors.toList());
    absolutes.forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

Wynikiem działania powyższego kodu będzie wydrukowanie na ekranie wszystkich ścieżek absolutnych do plików, które znajdują się w katalogu `zajavka` w obecnym **WD**.

lines()

Wcześniej wspominałem o metodzie `readAllLines()` i wspomniałem, że nie poleca się jej używać do dużych plików, ze względu na to, że cała zawartość pliku jest ładowana do pamięci jednocześnie. Istnieje natomiast taka metoda jak `Files.lines()`, która zwraca `Stream<String>` i obchodzi ten problem. Metoda ta ładuje do pamięci tylko fragment pliku, a nie jego całość.

```
Path path = Paths.get("zajavka/myFile.txt");
try {
    Stream<String> lines = Files.lines(path);
    List<String> mapped = lines
        .filter(line -> line.contains("2") || line.contains("4")) ①
        .map(String::toUpperCase) ②
        .collect(Collectors.toList());
    mapped.forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

① Zostaw tylko te linijki, które zawierają w sobie wartość `2` lub `4`.

② Zamień wszystkie pozostałe linijki na pisane z wielką literą.

Wynikiem działania powyższego kodu będzie wydrukowanie na ekranie linijek z pliku `myFile.txt`, który znajdują się w katalogu `zajavka` w obecnym **WD**. Zostaną wydrukowane tylko te linijki, które zawierają w sobie wartość `2` lub `4`. Zostaną one wydrukowane w całości wielką literą.

Porównanie starych i nowych metod

Stara metoda	NIO.2
file.getName()	path.getFileName()
file.getAbsolutePath()	path.toAbsolutePath()
file.exists()	Files.exists(path)
file.isDirectory()	Files.isDirectory(path)
file.isFile()	Files.isRegularFile(path)
file.isHidden()	Files.isHidden(path)
file.length()	Files.size(path)
file.lastModified()	Files.getLastModifiedTime(path)
file.setLastModified(time)	Files.setLastModifiedTime(path, time)
file.delete()	Files.delete(path)
file.renameTo(file)	Files.move(source, destination)
file.mkdir()	Files.createDirectory(path)
file.mkdirs()	Files.createDirectories(path)
file.listFiles()	Files.list(path)