Notatki - NIO.2 Paths

Spis treści

NIO.2, Path i Paths

I nie jest to przekręcona nazwa (NIO.2) - tak się to nazywa.

Może zaczynając trochę od historii zmian w API Javy w kontekście klas służących do operowania na plikach.

Paczka `java.io` została wprowadzona w Javie w wersji 1.0, czyli dawno temu. Były tam zawarte klasy umożliwiające operowanie na plikach przy wykorzystaniu pojedynczych bajtów, czyli klasy z nazwami Stream, których tematykę poruszaliśmy.

W Javie w wersji 1.4 zostały wprowadzone m.in. taki klasy jak `ByteBuffer` lub `Channels`, które z założenia miały poprawiać wydajność poprzednich rozwiązań. Klasy wprowadzone w wersji 1.4 wyróżniały się tym, że stosowały rozwiązania nieblokujące (non-blocking). Oznacza to, że poprzednie rozwiązania blokowały zasób na którym pracowały, czyli przykładowo plik na którym operowały. Od teraz możliwe było stosowanie tymczasowego bufora, który nie blokował zasobu, na którym pracujemy. Zbiorczo te rozwiązania są określane jako NIO (non-blocking IO) i dodane zostały w paczce `java.nio`. Nie

poruszaliśmy natomiast tematyki ich wykorzystania.

W Javie 7 zostały wprowadzone kolejne klasy, które określa się zbiorczo jako **NIO.2**. Gdzieś kiedyś przeczytałem taką ciekawostkę, że zmiany w Javie 1.4 miały zastąpić rozwiązania wprowadzone na samym początku, natomiast dopiero zmiany wprowadzone w Javie 7 faktycznie to zrobiły ☺.

W tej notatce poruszymy tematykę klas wprowadzonych w Javie 7. Dołożymy też do tego zmiany wprowadzone przez Javę 8, doszły wtedy metody umożliwiające funkcyjne podejście do tematu. Jak też z każdymi poprawkami, te poprawki również miały poprawić wydajność działania aplikacji ☺.

Jeżeli się zastanawiasz czemu poruszaliśmy tematykę Streamów, skoro teraz napisałem, że w Javie 7 wprowadzono lepsze rzeczy? Bo w praktyce nadal często można spotkać użycie wspomnianych Streamów. A druga kwestia to jest rozumienie sposobu działania pod spodem. Niedługo zostaną pokazane inne sposoby, których będziesz prawdopodobnie używać najczęściej, natomiast warto jest znać sposób działania mechanizmów "pod spodem". Często będzie to wyglądało w ten sam sposób w odniesieniu do innych zagadnień, czyli będziemy poruszać przykłady kodu, których nie będziesz praktycznie pisać w praktyce, ale bardzo ważne jest w takich przypadkach zrozumienie sposobu działania mechanizmu. Ważne jest to z tego powodu, że często mechanizmy działają do siebie analogicznie lub podobnie.

Dlaczego nie wspominam rzeczach wprowadzonych np. w Java 15, a skupiam się tylko wokół rzeczy wprowadzonych w Java 7 lub 8? Bo w nowszych wersjach nie zostały wprowadzone zmiany dotyczące tych zagadnień, które byłby istotne na tym etapie nauki.

Path

Do tych wszystkich klas, które już poznaliśmy dorzucimy jeszcze interfejs `java.nio.Path`. Można to rozumieć jako byt reprezentujący ścieżki w systemie operacyjnym razem z całą hierarchią tych ścieżek. Ścieżka taka może wskazywać na plik lub folder. Rozumienie było takie, aby interfejs `Path` zastąpił klasę `File`, którą poznaliśmy wcześniej. Wspominamy o obu z nich, bo obydwie spotyka się ciągle w użyciu.

Filozoficznie patrząc, interfejs `Path` może być używane zamiast `File` gdyż dostarcza on tych samych funkcjonalności, dodaje on również nową funkcjonalność. Wspiera dowiązania symboliczne (*symbolic links*, *symlink*). Są to takie pliki, które zawierają odniesienia do innego pliku lub katalogu w postaci ścieżki do tego pliku lub katalogu. Po co się to stosuje? Jeżeli przykładowo chcemy mieć w projekcie odniesienie do jakiegoś pliku, który może często zmieniać lokalizację. W takim przypadku możemy w konfiguracji projektu dodać lokalizację takiego symlinka i w symlinku zmieniać namiar na plik, który nas interesuje. Na ten moment żeby dalej nie komplikować, nie będziemy zagłębiać się w temat.

Jak stworzyć Path

Tutaj zaczyna się zabawa. W przypadku klasy `File`, wywoływaliśmy konstruktor podając namiar na plik na dysku i wszystko działało. Tutaj już tak nie jest ☺. Pierwsza rzecz jest taka, że `Path` jest interfejsem, więc nie ma konstruktora, żeby stworzyć instancję. Pojawia się zatem pytanie, czemu `Path` jest interfejsem. W nowym podejściu (czyli tym NIO.2), utworzenie w Javie obiektu reprezentującego folder lub plik jest operacją zależną od systemu plików, co wynika z systemu operacyjnego, na którym pracujemy. Czyli `Path` jest tylko interfejsem, który zapewnia nam dostęp do metod, natomiast konkretna

implementacja zależna od systemu plików musi nam zostać dostarczona w inny sposób.

I wtedy na białym koniu wjeżdża klasa `java.nio.files.Paths`, która ma zdefiniowane metody statyczne, pozwalające na utworzenie obiektu implementującego interfejs `Path`. Jednocześnie robi to w taki sposób, że logika pod spodem jest zależna od systemu plików, jaki funkcjonuje w danym systemie operacyjnym. Natomiast z perspektywy programisty piszącego kod, możemy używać metod z klasy `Paths` i dla nas jest to przeźroczyste.

Przykład użycia w kodzie:

```
Path path1 = Paths.get("src/pl/zajavka/java/nio/myFile.txt");

// Druga opcja jest o tyle dobra, że nie musimy się zastanawiać nad separatorem
Path path2 = Paths.get("src", "pl", "zajavka", "java", "nio", "myFile.txt");
Path path3 = Paths.get("nio", "myFile.txt");

// Nie ma metody path.exists() tak jak to było w file
System.out.println("path1: " + path1 + " exists: " + Files.exists(path1));
System.out.println("path2: " + path2 + " exists: " + Files.exists(path2));
System.out.println("path3: " + path3 + " exists: " + Files.exists(path3));
```

Już widzisz, że mamy takie klasy/interfejsy jak `Path`, `Paths`, `File`, `Files`. O klasie `Files` napiszemy później. Można natomiast powiedzieć, że `Paths` i `Files` są takimi klasami 'pomocniczymi', które zawierają metody statyczne aby coś zrobić. Tak jak w tym przypadku, przy wykorzytaniu klasy `Files` nastąpiło sprawdzenie, czy plik pod podaną ścieżką istnieje.

URI

Skoro mówimy o `Path`, należy też wspomnieć o `java.net.URI`. Jedna z metod `Paths.get()` przyjmuje obiekt `URI` jako argument. `URI` (*Uniform Resource Identifier*) jest ustandaryzowanym sposobem na identyfikację jakiegoś zasobu. Jego zapis rozpoczyna się od schematu, przykładowo `http://`, `https://`, `file://`, a następnie zawiera lokalizację zasobu.

Wiedząc już o jego istnieniu, można napisać kod pokazany poprzednio w taki sposób:

```
try {
    Path path1 = Paths.get(new URI("src/pl/zajavka/java/nio/myFile.txt"));
} catch (Exception e) {
    System.err.println("path1 missing scheme: " + e.getMessage());
}
```

Trzeba natomiast pamiętać, że używając `URI` należy podać schemat zasobu, zatem przykład wyżej jest nieprawidłowy. Poniżej jest prawidłowy.

```
try {
    Path path2 = Paths.get(new URI("file://src/pl/zajavka/java/nio/myFile.txt"));
} catch (Exception e) {
    System.out.println("won't print here");
}
```

FileSystem

Dostępna jest również klasa `java.nio.file.FileSystem`, która jest w stanie pokazać nam jaki obiekt systemu plików faktycznie jest tworzony na naszej maszynie. Możemy napisać taki kod:

```
FileSystem fileSystem = FileSystems.getDefault();
System.out.println(fileSystem);
```

W moim przypadku drukuje się coś podobnego do:

```
sun.nio.fs.WindowsFileSystem@3a03464
```

Korzystając teraz ze zmiennej `fileSystem` możemy wywołać:

```
Path path = fileSystem.getPath("src/pl/zajavka/java/nio/myFile.txt");
```

Co da nam ten sam efekt, co wywołanie `Paths.get()`.

Konwersja między Path a File

Jeżeli natomiast chcielibyśmy przejść z używania klasy `Path` na `File` i odwrotnie, można to zrobić w ten sposób:

```
File file1 = new File("someFile.txt");
Path path1 = file1.toPath();

Path path2 = Paths.get("someFile.txt");
File file2 = path2.toFile();
```

Operacje na obiektach typu Path

Tak sobie piszemy o tym interfejsie `Path` i nie napisaliśmy jednej najważniejszej rzeczy, o której trzeba pamiętać. Jeżeli stworzymy obiekt `Path`, to nie jest to plik, który istnieje na dysku, tylko reprezentacja lokalizacji pliku w systemie operacyjnym. Taki plik wcale nie musi fizycznie istnieć na dysku, żeby móc operować na obiekcie `Path`. `Path` dostarcza nam natomiast metody, które wymagają aby plik faktycznie się na tym dysku znajdował, inaczej dostaniemy wyjątek.

Przejdźmy zatem do omówienia kolejnych metod.

toString()

Zwraca reprezentację `Path` jako `String`. Standardowa metoda z klasy `Object`. Jest to jedna z niewielu metod z Path, która zwraca `Stringa`. Uczulam na to, bo inne metody zwracają `Path`.

```
Path path = Paths.get("C:/zajavka/someFile.txt");
String toString = path.toString();
System.out.println(toString); ①
```

① Wydrukowane zostanie: C:\zajavka\someFile.txt

getNameCount()

Ta metoda jest używana aby otrzymać ilość "elementów" w ścieżce, czyli w praktyce zlicza nam ilość folderów w podanej ścieżce razem z plikiem, który może wystąpić na końcu ścieżki.

```
Path path = Paths.get("C:/zajavka/someFile.txt");
int nameCount = path.getNameCount();
System.out.println(nameCount); ①
```

① Wydrukowane zostanie: 2

getName(int index)

Ta metoda zwraca nam nowy obiekt typu `Path`, który występuje na miejscu podanego indeksu w oryginalnym obiekcie `Path`. Warto zwrócić uwagę, że indeksowanie jest od `0`. Kolejna ważna rzecz, to to, że `path.getName(0)` nie zwraca nam roota, czyli w przypadku np. Windowsa, nie dostaniemy np. Dysku `C`, tylko pierwszy katalog na ścieżce.

```
Path path = Paths.get("C:/zajavka/someFile.txt");
Path name0 = path.getName(0);
System.out.println(name0); ①
```

① Wydrukowane zostanie: zajavka

getFileName()

Metoda zwraca nam nazwę pliku, który jest wskazany na ścieżce. Plik jest rozumiany jako najdalszy element od początku ścieżki, czyli w przypadku braku zdefiniowanego pliku w ścieżce (gdy ścieżka zawiera same katalogi), metoda ta zwróci ostatni katalog.

```
Path path = Paths.get("C:/zajavka/someFile.txt");
Path fileName = path.getFileName();
System.out.println(fileName); ①

Path path1 = Paths.get("C:/zajavka/someCatalog");
Path fileName1 = path1.getFileName();
System.out.println(fileName1); ②
```

① Wydrukowane zostanie: someFile.txt

② Wydrukowane zostanie: someCatalog

toAbsolutePath()

Metoda zwraca nam obiekt typu `Path` reprezentujący absolutną ścieżkę utworzoną na podstawie ścieżki, którą podaliśmy. Jeżeli podajemy ścieżkę relatywną, wywołanie `toAbsolutePath()` stworzy nam ścieżkę absolutną na podstawie miejsca, w którym uruchamiany jest projekt w systemie operacyjnym.

```
Path path = Paths.get("C:/zajavka/someFile.txt");
Path absolutePath = path.toAbsolutePath();
System.out.println(absolutePath); ①

Path path1 = Paths.get("zajavka/someFile.txt");
Path absolutePath1 = path1.toAbsolutePath();
System.out.println(absolutePath1); ②
```

① Wydrukowane zostanie: C:\zajavka\someFile.txt

② Wydrukowane zostanie: C:\Users\krogowski\zajavka\someFile.txt

isAbsolute()

Metoda zwraca `boolean` mówiący czy podana ścieżka jest absolutna czy nie.

```
Path path = Paths.get("C:/zajavka/someFile.txt");
boolean absolute = path.isAbsolute();
System.out.println(absolute); ①

Path path1 = Paths.get("zajavka/someFile.txt");
boolean absolute1 = path1.isAbsolute();
System.out.println(absolute1); ②
```

① Wydrukowane zostanie: true

② Wydrukowane zostanie: false

getParent()

Metoda zwraca `Path` reprezentujący rodzica ostatniego pliku lub katalogu na ścieżce. Przykładowo:

```
Path path = Paths.get("C:/zajavka/someDirectory/someFile.txt");
Path parent = path.getParent();
System.out.println(parent); ①

Path path1 = Paths.get("zajavka/someDirectory/someFile.txt");
Path parent1 = path1.getParent();
System.out.println(parent1); ②
```

① Wydrukowane zostanie: C:\zajavka\someDirectory

② Wydrukowane zostanie: zajavka\someDirectory

getRoot()

Pobierz korzeń... Metoda ta zwraca katalog główny (katalog root), czyli katalog nadrzędny dla

wszystkich innych katalogów. Jeżeli ścieżka jest relatywna, zostanie zwrócony `null`.

```
Path path = Paths.get("C:/zajavka/someDirectory/someFile.txt");
Path root = path.getRoot();
System.out.println(root); ①

Path path1 = Paths.get("zajavka/someDirectory/someFile.txt");
Path root1 = path1.getRoot();
System.out.println(root1); ②
```

① Wydrukowane zostanie: C:\

② Wydrukowane zostanie: null

subpath()

Metoda ta analogicznie jak `substring()` służy do stworzenia podścieżki na podstawie przekazanych indeksów ścieżki oryginalnej. Pierwszy argument jest inkluzywny, drugi nieinkluzywny (ekskluzywny, znaczy taki premium? 😄).

```
Path path = Paths.get("C:/zajavka/someDirectory/anotherDirectory/yetAnother/someFile.txt");
Path subpath1 = path.subpath(1, 3);
Path subpath2 = path.subpath(2, 4);
Path subpath3 = path.subpath(0, 2);
System.out.println(subpath1); ①
System.out.println(subpath2); ②
System.out.println(subpath3); ③

Path path1 = Paths.get("zajavka/someDirectory/anotherDirectory/yetAnother/someFile.txt");
Path subpath4 = path1.subpath(1, 3);
Path subpath5 = path1.subpath(2, 4);
Path subpath6 = path1.subpath(0, 2);
System.out.println(subpath4); ④
System.out.println(subpath5); ⑤
System.out.println(subpath6); ⑥
```

① Wydrukowane zostanie: someDirectory\anotherDirectory

② Wydrukowane zostanie: anotherDirectory\yetAnother

③ Wydrukowane zostanie: zajavka\someDirectory

④ Wydrukowane zostanie: someDirectory\anotherDirectory

⑤ Wydrukowane zostanie: anotherDirectory\yetAnother

⑥ Wydrukowane zostanie: zajavka\someDirectory

relativize()

Metoda ta służy do skonstruowania relatywnej ścieżki ze ścieżki pierwszej do ścieżki drugiej. Innymi słowy, efektem jest ścieżka, po której przejściu ze ścieżki pierwszej, znajdziemy się w ścieżce drugiej.

```
Path path1 = Paths.get("E:/zajavka/someDirectory/anotherDirectory/yetAnother/someFile.txt");
Path path2 = Paths.get("E:/zajavka/someDirectory/anotherDirectory/secondDirectory/anotherFile.txt");
System.out.println(path1.relativize(path2)); ①
System.out.println(path2.relativize(path1)); ②

Path path3 = Paths.get("someFile.txt");
Path path4 = Paths.get("anotherFile.txt");
System.out.println(path3.relativize(path4)); ③
System.out.println(path4.relativize(path3)); ④
```

① Wydrukowane zostanie: ..\..\secondDirectory\anotherFile.txt

② Wydrukowane zostanie: ..\..\yetAnother\someFile.txt

③ Wydrukowane zostanie: ..\anotherFile.txt

④ Wydrukowane zostanie: ..\someFile.txt

Przykładach `path3` i `path4` pojawiają się `..` gdyż aby przejść z pliku `someFile.txt` do `anotherFile.txt` musimy najpierw dostać się do katalogu rodzica. W odwrotną stronę działa to identycznie.

Jeżeli mamy do czynienia z 2 ścieżkami absolutnymi jak w podanym przypadku, Java nie wykonuje sprawdzenia czy ścieżki te faktycznie istnieją na dysku, stąd też podałem ścieżki na dysku `E`, który na mojej maszynie nie istnieje.

resolve()

A może chcemy dodać do siebie 2 ścieżki? To też tak można.

```
Path path5 = Paths.get("zajavka/../someDirectory/anotherDirectory");
Path path6 = Paths.get("secondDirectory/../anotherCatalog");
System.out.println(path5.resolve(path6)); ①
```

① Wydrukowane zostanie: zajavka\.\someDirectory\anotherDirectory\secondDirectory\..\anotherCatalog_

Trzeba tutaj natomiast zwrócić uwagę, że ta metoda "nie sprząta" ścieżek. Tzn, że jeżeli na ścieżce występują znaki `..`, to ta metoda tego nie sprzątnie, tylko przeklei dalej tak jak było. Od sprzątania jest następna metoda.

normalize()

Tą metodą możemy posprzątać naszą ścieżkę. Czyli jeżeli na naszej ścieżce występowały `..`, to `normalize()` skróci tę ścieżkę w taki sposób, aby wynik końcowy był taki sam jak przed wywołaniem tej metody, ale zapis będzie krótszy.

```
Path path7 = Paths.get("zajavka/../someDirectory/anotherDirectory/secondDirectory/../anotherCatalog");
System.out.println(path7.normalize()); ①
```

① Wydrukowane zostanie: someDirectory\anotherDirectory\anotherCatalog

Tutaj znowu należy zaznaczyć, że Java nie sprawdza czy podana ścieżka faktycznie występuje na dysku.

toRealPath()

I ostatnia metoda jaką poruszaliśmy. Ta metoda zachowuje się trochę jak `toAbsolutePath()`, przy czym sprawdza jednocześnie czy plik faktycznie istnieje na dysku. Jeżeli nie istnieje, to wyrzuca znany nam już `NoSuchFileException`.

Warto też wiedzieć, że jeżeli mamy zdefiniowaną naszą ścieżkę jako nieznormalizowaną (czyli np. z `..`) to ta metoda pod spodem dokona normalizacji takiej ścieżki.

```
Path path1 = Paths.get("zajavka/someDirectory/anotherDirectory/yetAnother/someFile.txt");
try {
    System.out.println(path1.toRealPath());
} catch (IOException e) { ①
    e.printStackTrace();
}
```

① Jeżeli plik nie istnieje, zostanie wyrzucone `java.nio.file.NoSuchFileException`

Working directory

I na koniec, znając już te wszystkie rzeczy, warto jest dodać dosyć ważną kwestię. Koncepcję czegoś takiego jak **working directory** - które jest używane gdy pracujemy na ścieżkach relatywnych. Jeżeli w naszej aplikacji będziemy podawać ścieżkę relatywną, to będzie ona użyta w kontekście naszego **working directory**, czyli jakby katalogu w którym odbywa się cała praca. Jak ustalić namiar na taki katalog?

Jeżeli spróbujemy odpalić przykłady takie jak np:

```
Path relative1 = Paths.get("java/nio_path/myFile.txt");

try {
    System.out.println("relative1.toRealPath(): " + relative1.toRealPath());
} catch (IOException e) {
    System.err.println("relative1 error: " + e.getMessage());
}
```

lub

```
Path relative1 = Paths.get("src/java/nio_path/myFile.txt");

try {
    System.out.println("relative1.toRealPath(): " + relative1.toRealPath());
} catch (IOException e) {
    System.err.println("relative1 error: " + e.getMessage());
}
```

To zobaczymy, że wywołując `toRealPath()`, Java zawsze postara się naszą ścieżkę relatywną dokleić do obecnego **working directory** (dodam, że każda aplikacja będzie miała swoje **WD**) i w konsekwencji wydrukowane zostanie coś takiego:

```
relative1 error: src/java/nio_path/myFile.txt
```

Przy czym moim **WD** jest:

```
C:\Users\krogowski\zajavka
```

Czyli Java próbuje naszą ścieżkę relatywną rozwiązać w kontekście swojego **WD**. Jak natomiast można sprawdzić jakie mamy **WD**?

```
try {
    System.out.println("Current WD: " + Paths.get(".").toRealPath()); ①
} catch (IOException e) {
    System.err.println("relative3 error: " + e.getMessage());
}
```

① Wydrukowane zostanie: Current WD: C:\Users\krogowski\zajavka