Notatki - System plików

Spis treści

File system . . . 1
- Plik . . . 1
- Katalog . . . 1
- File system . . . 1
- Ścieżka . . . 2
- Absolute path . . . 2
- Relative path . . . 2
- Separator . . . 2
- Symbole charakterystyczne . . . 3

Klasa File . . . 3
- Użycie obiektu klasy File . . . 4
- Popularne metody z klasy File . . . 4

File system

Zanim wejdziemy w temat plików, powiedzmy sobie czym jest plik, a czym jest katalog.

Plik

Plik (*file*) jest zbiorem danych, który przechowuje informacje użytkownika albo systemu operacyjnego. Plik zawiera w sobie skończoną ilość danych oraz posiada atrybuty, takie jak np. rozmiar, data utworzenia, data modyfikacji. Pliki są umiejscowione w katalogach (folderach).

Katalog

Katalog (*directory*) jest bytem w systemie operacyjnym, który może zawierać pliki albo inne foldery. W kodzie, do folderów często odwołujemy się identycznie jak do plików, wynika to z tego, że mają one podobne atrybuty, jak chociażby data utworzenia. Należy również pamiętać, że zarówno pliki jak i foldery znajdujące się w tym samym folderze muszą mieć unikalną nazwę.

W nomenklaturze istnieje takie pojęcie jak **root**. Jest to najwyższy folder, do którego możemy się odwołać. W przypadku Windowsa, może to być `C:\`.

File system

System plików (*file system*) - jest on odpowiedzialny za odczyt i zapis plików na komputerze. Różne systemy operacyjne korzystają z różnych systemów plików. Klasycznym przykładem jest `Windows` i `Linux`, które korzystają z różnych systemów plików. Całe szczęście Java dostarcza nam API, dzięki któremu możemy operować na plikach w różnych systemach plików. Pozwala nam to na wykorzystanie tego

samego kodu do operowania na różnych systemach operacyjnych.

Nie zagłębiamy się w różnice pomiędzy systemami plików używanych w konkretnych systemach operacyjnych. Ważne jest natomiast dla nas to, że jak napiszemy kod w Javie, to może on być uruchomiony zarówno na `Windows` jak i na `Linux`, bo Java poradzi sobie z tymi systemami plików.

Ścieżka

Ścieżka (*path*) jest reprezentacją pliku albo katalogu w systemie plików w taki sposób, że wiemy konkretnie do jakiego pliku lub katalogu się odnosimy. Coś jak adres mieszkania. W Windowsie ścieżki są zapisywane z backslashem `\`, a na Linux ze zwykłym slashem `/`.

Mieliśmy już z tym wszystkim do czynienia wcześniej, podczas gdy tworzyliśmy klasy i paczki w Javie. Tworząc klasę, Intellij tworzy nam na dysku plik `NazwaKlasy.java`, tworząc paczkę natomiast, tworzymy na dysku katalog z nazwą paczki.

Absolute path

Ścieżka absolutna to pełna ścieżka do pliku zaczynająca się od roota, uwzględniając wszystkie katalogi, które znajdują się po drodze od roota do naszego pliku lub katalogu.

Przykład ścieżki absolutnej

```
C:\Users\zajavka\work\bootcamp\Main.java
```

Relative path

Ścieżka relatywna to ścieżka do naszego pliku, relatywnie odnosząca się do innego katalogu.

Przykład ścieżki relatywnej

```
bootcamp\Main.java
```

Ścieżka taka może tak na prawdę odnosić się do kilku różnych miejsc, bo `bootcamp\Main.java` może istnieć w wielu katalogach na dysku. W przypadku ścieżki absolutnej, mówimy np. o jednym konkretnym unikalnym pliku.

Separator

Przypomnijmy sobie co zostało wspomniane wcześniej. `Windows` i `Linux` mają inne separatory katalogów/plików. W Windows ścieżki są zapisywane z backslashem `\`, a na Linux ze zwykłym slashem `/`.

Java może nam podać informację, co jest separatorem w sposób pokazany poniżej:

```
// Można tak
System.out.println(System.getProperty("file.separator"));
// Albo tak
System.out.println(java.io.File.separator);
```

W programie Javowym do zapisu ścieżki, możemy używać separatora `/`. Java sobie dalej z tym poradzi, nawet gdy używamy Windowsa, w którym separatorem jest `\`.

Symbole charakterystyczne

Ogólnie jest przyjęte stosowanie 2 charakterystycznych symboli na ścieżkach do plików.

- **.** - (pojedyncza kropka) - oznacza odniesienie do katalogu, w którym znajdujemy się obecnie
- **..** - (podwójna kropka) - oznacza odniesienie do katalogu rodzica, czyli katalogu, w którym znajduje się nasz katalog

Czyli poniższy przykład:

```
../someDirectory/someFile.txt
```

Oznacza, że z katalogu, w którym znajdujemy się obecnie, wyjdziemy do katalogu rodzica (do katalogu wyżej) i tam poszukamy folderu `someDirectory` z plikiem `someFile.txt`.

Możemy też zapisywać bardziej złożone kombinacje, np:

```
../../../someFile.txt
```

Co oznacza, że 3 razy staramy się wyjść "do góry" w stosunku do katalogu, w którym znajdujemy się obecnie i tam poszukamy pliku `someFile.txt`

Natomiast taki zapis:

```
./someFile.txt
```

Oznacza, że odnosimy się do pliku `someFile.txt` w katalogu, w którym znajdujemy się obecnie.

Klasa File

Bardzo często będziemy używać klasy `java.io.File`, dlatego to od niej zaczniemy. Klasa ta jest używana do odczytywania informacji o plikach, tworzenia lub usuwania plików. Jednocześnie klasa ta nie może modyfikować zawartości pliku - do tego są inne sposoby.

Jest to dosyć nieintuicyjne dlatego chcemy to zaznaczyć. Klasa `File` może operować zarówno na plikach jak i katalogach.

Użycie obiektu klasy File

```
public class FileExamples {

    public static void main(String[] args) {
        File file1 = new File("myFileInDir.txt");
        System.out.println(file1.exists());
        File file2 = new File("myFolder/myFileInDir.txt");
        System.out.println(file2.exists());
        File file3 = new File("myAnotherFolder/myFolder/myFileInDir.txt");
        System.out.println(file3.exists());
    }
}
```

Gdy uruchamiamy nasz program z poziomu Intellij, możemy zdefiniować ścieżkę relatywną i plik zostaje znaleziony - pod warunkiem, że ścieżka relatywna zaczyna się od roota naszego projektu. Dlatego z jednej strony bezpieczniej jest posługiwać się ścieżką absolutną. Ale gdy taki kod jest uruchamiany na różnych komputerach, to przecież ta ścieżka absolutna może być inna. Dlatego najczęściej używa się ścieżek relatywnych w odniesieniu do lokalizacji uruchamianego programu.

Pamiętaj - obiekt `File` w Javie możemy utworzyć gdy plik fizycznie nie istnieje na dysku. Możemy taki obiekt również utworzyć, gdy taki plik na dysku istnieje.

Popularne metody z klasy File

Nazwa metody	Opis
exists()	Sprawdza, czy plik istnieje
isDirectory()	Sprawdza czy obiekt File jest katalogiem
isFile()	Sprawdza czy obiekt File jest plikiem
length()	Zwraca ilość bajtów w pliku. Może wystąpić taka sytuacja, że plik zarezerwuje na dysku więcej bajtów niż faktycznie jest potrzebne, kwestia performance
lastModified()	Zwraca ilość milisekund od Epoch do momentu, gdy plik był edytowany po raz ostatni
delete()	Usuwa z dysku plik albo katalog. Aby usunąć katalog, musi on być pusty
renameTo(File)	Zmienia nazwę pliku na podaną jako argument
mkdir()	Tworzy katalog, pod ścieżką określoną w obiekcie File
mkdirs()	Jeżeli mamy na podanej ścieżce kilka nieistniejących katalogów, zostaną stworzone wszystkie nieistniejące
getName()	Zwraca nazwę pliku
getAbsolutePath()	Zwraca ścieżkę absolutną
getParent()	Zwraca ścieżkę rodzica albo null jeżeli takiego rodzica nie ma
listFiles()	Zwraca tablicę File[] z plikami lub katalogami w danym katalogu

Poniżej przykład wykorzystania niektórych metod z tabeli:

```
public class FileExamples {

    void example() {
        File file = new File("someFile.txt");
        if (!file.exists()) {
            System.out.println("File: " + file.toString() + " doesn't exist");
            return;
        }

        System.out.println("File: " + file.toString() + " exists");
        System.out.println("file.getAbsolutePath(): " + file.getAbsolutePath());
        System.out.println("file.getParent(): " + file.getParent());

        if (file.isFile()) {
            System.out.println("File: " + file.toString() + " is file");
            System.out.println("file.length(): " + file.length());
            return;
        }

        if (file.isDirectory()) {
            System.out.println("File: " + file.toString() + " is directory");
            for (File subfile : file.listFiles()) {
                System.out.println("subfile.getName(): " + subfile.getName());
            }
        }
    }
}
```