Notatki - Streamy - FileInputStream oraz FileOutputStream

Spis treści

FileInputStream oraz FileOutputStream

Wymienione klasy (`FileInputStream` oraz `FileOutputStream`) są najbardziej podstawowymi, o których będziemy rozmawiać. Służą do odczytu bajtów z pliku oraz do zapisu bajtów do pliku. Odczyt z pliku następuje aż do odczytania wartości `-1`. Gdy napotkamy taką wartość, należy przerwać odczyt z pliku, oznacza to jego koniec.

Przykład w kodzie:

Klasa StreamsExamples

```
public class StreamsExamples {

    public static void main(String[] args) throws IOException {
        File inputFile = new File("src/pl/zajavka/java/myInputFile.txt");
        File outputFile = new File("src/pl/zajavka/java/myOutputFile.txt");
        CopyExample.justCopyNoBuffer(inputFile, outputFile);
    }

}
```

Klasa CopyExample

```
public class CopyExample {

    static void justCopyNoBuffer(File source, File destination) throws IOException {
        try (
            InputStream input = new FileInputStream(source);
            OutputStream output = new FileOutputStream(destination)
        ) {
            int value = input.read();
            System.out.printf("Starting reading file: [%s]%n", source);
            System.out.printf("Read value: [%s], char: [%s]%n", value, (char) value);

            while (value != -1) {
                System.out.printf("Writing value: [%s], char: [%s]%n", value, (char) value);
                output.write(value);
                value = input.read();
                System.out.printf("Read value: [%s], char: [%s]%n", value, (char) value);
            }
        }
    }
}
```

Jeżeli chcemy zobaczyć jakie znaki stoją pod wczytanymi bajtami, możemy odwołać się do tablicy `Unicode` (zaraz zostanie opisane, że nie zawsze to zadziała). Możemy wtedy próbować rzutować otrzymaną wartość liczbową na `char` żeby zobaczyć jaki to znak (sposób ten działa dla np. angielskich liter, więcej poniżej).

Pamiętać również należy, że przykład powyżej jest wolny pod względem wydajności bo operujemy na pojedynczych wartościach bajtów.

Sytuacja opisana powyżej będzie działała poprawnie np. dla **nie**polskich znaków. Jeżeli chcemy wygodnie operować na polskich znakach musimy przejść do klas "...Reader" oraz "...Writer" i zastosować wspomniane wcześniej kodowanie znaków.

Po przeanalizowaniu kodu powyżej zwróć uwagę, że odczytywane wartości są typu `int` a nie `byte`. Wynika to z tego, że zakres `byte` to (-128;127), podczas gdy my potrzebujemy odczytywać wartości w zakresie `(0;255)`. Druga sprawa jest taka, że używając typu `byte` pozbawiamy się możliwości oznaczenia `-1` reprezentującej koniec pliku (`EOF` - end of file), gdyż `-1` jest poprawną wartością typu `byte`. Dlatego też, odczytywane wartości są typu `int`.

Należy do tego wszystkiego dodać jeszcze dwa terminy:

- signed variables (zmienne ze znakiem) - oznacza wartości, które mogą być reprezentowane zarówno przez liczby dodatnie jak i ujemne. Możemy zatem określić stwierdzenia takie jak: `signed byte` lub `signed int`.
- unsigned variables (zmienne bez znaku) - oznacza wartości, które mogę być reprezentowane tylko przez liczby dodatnie oraz zero. Możemy zatem określić stwierdzenia takie jak: `unsigned byte` lub `unsigned int`.

Dlaczego ma to tutaj znaczenie? Wcześniej zostało napisane, że w kodzie powyżej wczytywany jest `int` a nie `byte` oraz wyjaśnione dlaczego. Jeżeli natomiast spróbujemy wykonać kod poniżej:

```
byte[] bytes = "ą".getBytes(StandardCharsets.UTF_8);
System.out.println(Arrays.toString(bytes));
```

To na ekranie zostanie wydrukowane `[-60, -123]`.

Dlaczego tak kręcę z tymi przykładami? Kodowanie znaków `UTF-8` może wykorzystać od 1 do 4 bajtów żeby zakodować jakiś znak. Na przykładzie literki `ą` widać, że wykorzystane zostały 2 bajty w celu jej zakodowania. Kodowanie znaków `UTF-8` pozwala nam określić, że dane dwa bajty mają reprezentować literkę `ą`. Jeżeli tę samą literkę `ą` zapiszemy w pliku tekstowym i spróbujemy ją wczytać stosując kod metody `justCopyNoBuffer()` to okaże się, że dostajemy wartości `196` oraz `133`. Metoda `justCopyNoBuffer()` z powodzeniem zapisze do innego pliku tę samą treść i będzie się ona poprawnie wyświetlała po zapisie w pliku, natomiast jak spróbujemy rzutować podane dwie wartości po ich wczytaniu na `char` to zgodnie z tablicą `Unicode` dostaniemy jakieś krzaki. Z tego można wyciągnąć dwa wnioski:

- `FileInputStream` oraz `FileOutputStream` po prostu czyta bajty, nie zastanawia się co dalej z tymi bajtami zrobić. Dopiero kodowanie `UTF-8` pozwala nam zinterpretować, że wczytane dwa bajty mają reprezentować literkę `ą`. Dlatego o tym wspominam, żeby później móc się do tego odwołać w przykładach z klasami "...Reader" oraz "...Writer".
- Przy odczycie danych za pomocą metody `read()` wartości wczytanych z pliku bajtów zostały zamienione na zakres (0;255). Zatem przy rzutowaniu tych wartości na `char` dostajemy dwa znaki stojące pod wartościami `196` oraz `133` zamiast jednego `ą`. Jeżeli chcielibyśmy, żeby te dwie wartości miały sensowne dla nas znaczenie należy zastosować kodowanie `UTF-8` i w tym celu trzeba użyć klas "...Reader" oraz "...Writer".

Kodowanie `UTF-8` było podawane jako przykład, przypominam, że nie jest to jedyny możliwy sposób kodowania.

Nie skupiamy się na algorytmie zamiany `signed byte` na `unsigned byte`. Jeżeli ktoś jest tym zainteresowany, zachęcam do pogrzebania w internecie 😀.