Notatki - Streamy - Teoria

Spis treści

Streamy . . . 1
- Ale Streamy już przecież były... . . . 1
- To czym są te Streamy tutaj? . . . 1
- Różnice między Streamami bajtowymi a znakowymi . . . 2
  - Character encoding . . . 3
  - ASCII i Unicode . . . 3
  - Unicode vs UTF-8 . . . 4
- Jak możemy sobie podzielić Streamy na kategorie? . . . 4

Jak żyć z tymi Streamami? . . . 7
- Otwieranie i zamykanie . . . 7
- Spuszczanie wody . . . 7
- Pomijanie . . . 8

Streamy

Ale Streamy już przecież były...

To nie są te same streamy co w programowaniu funkcyjnym, pomimo tego, że nazwy są mylące ☺. Pod tym linkiem znajdziesz przykładową odpowiedź ze Stackoverflow wyjaśniającą tę kwestię. Możemy w niej przeczytać, że pechowo wyszło, że zarówno Streamy IO oraz Streamy, które poznaliśmy w programowaniu funkcyjnym nazywają się tak samo. IO Streamy (czyli te, o których zaraz będziemy rozmawiać) służą do operowania na zewnętrznych zasobach (np. plikach na dysku). Streamy w programowaniu funkcyjnym służą do przetwarzania danych w sekwencji. Koncepcje te należy rozumieć niezależnie od siebie.

Możliwe jest jednak używanie tych mechanizmów razem, np. klasa `BufferedReader` posiada zdefiniowane metody, które mogą zwracać `Stream<String>`, ale do tego przejdziemy później.

Podobieństwo jakie możemy wyróżnić pomiędzy IO Stream oraz "functional" Stream to abstrakcja, dzięki której możemy mówić o sekwencji danych. Zarówno jedna i druga koncepcja Streamów odnosi się do sekwencji danych, która to sekwencja ma jakiś początek i jakiś koniec przetwarzania.

Przykładowa różnica jest taka, że IO Streamy pozwalają nam tylko odczytywać i zapisywać dane (np. z pliku). IO Streamy nie mają wbudowanego mechanizmu do przetwarzania tych danych, jak "functional" Stream i metody `map()` lub `filter()`.

To czym są te Streamy tutaj?

Zawartość plików może być odczytywana lub zapisywana przy wykorzystaniu IO Streamów (In/Out).

Streamy są strumieniem danych, z którego możesz odczytywać lub do którego możesz zapisywać informacje.

Stream jest strukturą danych, którą można rozumieć jako stały przepływ informacji. Możemy sobie wyobrazić to jako przepływ wody, gdzie kropla po kropli mamy umieszczone jakieś informacje. Istnieją Streamy, które mogą przetwarzać dane kropla po kropli. Są też takie, które mogą przetwarzać kilka kropli jednocześnie (mają wtedy w nazwie `Buffered`).

Streamy nie mają indeksów jak przykładowo w tablicach, nie możemy również przejść w takim Streamie do przodu ani do tyłu, tak jakbyśmy przykładowo mogli w tablicach.

Streamy są typowo oparte o bajty albo znaki. Streamy, które są oparte na bajtach najczęściej mają w swojej nazwie fragment "...Stream". Niedługo poznamy `InputStream` oraz `OutputStream`. Streamy tego typu zapisują oraz odczytują surowe bajty. Streamy oparte o znaki nazywają się najczęściej "...Reader" lub "...Writer".

Pomocne w wyobrażeniu sobie Streamów może być spojrzenie na problem zważenia bardzo ciężkiego worka z piaskiem. Dodajmy do tego, że waga nie jest w stanie zważyć całego worka jednocześnie. Możemy go wtedy podzielić na mniejsze worki przesypując piasek i ważyć je oddzielnie. Czyli dzielimy worek na porcje i procesujemy kolejne porcje.

Pomimo, że wspomnieliśmy, że występują Streamy bajtowe i znakowe, w praktyce pod spodem praktycznie wszystkie Streamy operują na bajtach. Sprowadza się to do odczytu lub zapisu pojedynczych bajtów, ewentualnie tablicy bajtów. Wokół tego zachowania zostały stworzone klasy, które są w stanie zapisywać więcej niż pojedyncze bajty ze względów wydajnościowych. W praktyce dużo czasu zajmuje komunikacja między aplikacją Javową a systemem plików, zatem zapisywanie lub odczytywanie pojedynczych bajtów może się stać niewydajne. Stąd klasy, które starają się zapisywać lub odczytywać wiele bajtów na raz.

Pomimo, że mówimy o Streamach w kontekście odczytu i zapisu plików z dysku, możliwe jest też ich używanie w kontekście komunikacji z innymi serwerami (czyli przez sieć).

Różnice między Streamami bajtowymi a znakowymi

Jeżeli klasa ma w nazwie Stream, operuje ona wtedy na bajtach. Jeżeli klasa ma w nazwie `Reader` lub `Writer`, oznacza to, że operacje odbywają się na znakach. Pomimo występowania nazewnictwa `Reader` lub `Writer`, nadal są to Streamy.

Pojawia się zatem pytanie, po co stosować Streamy znakowe, skoro można zawsze używać Streamów bajtowych?

Są one dostosowane do typowych operacji tekstowych, zatem ułatwiają operowanie kodem jeżeli chcemy manipulować danymi tekstowymi. Druga rzecz to kodowanie plików. I znowu pojawia się nowy termin, czym jest kodowanie?

Termin "kodowanie plików" lub "kodowanie znaków" nie jest aż taki nowy, pojawiał się już parę razy, ale wyjaśnijmy go sobie raz na zawsze 😀.

Character encoding

Kodowanie znaków określa w jaki sposób znaki są kodowane i przechowywane w reprezentacji bajtowej i w jaki sposób z bajtów są one dekodowane z powrotem jako znaki. Mamy dostępnych dużo różnych rodzajów kodowań i zwyczajnie określają one jaki znak tekstowy ma być reprezentowany przez podaną wartość bajta.

Jeżeli ktoś byłby zainteresowany zabawą różnymi rodzajami kodowań znaków, to polecam pobawić się poniższym kodem:

```
// Drukujemy na ekranie domyślne kodowanie
System.out.println("Charset.defaultCharset: " + Charset.defaultCharset());

// Zabawa będzie widoczna dopiero na polskich znakach,
// bo to właśnie znaki diaktryczne stanowią najczęściej problem
String sentence = "some sentence to work on ąęół";
byte[] bytes = sentence.getBytes(StandardCharsets.UTF_8);
System.out.println(Arrays.toString(bytes));
String result = new String(bytes, StandardCharsets.UTF_8);
System.out.println(result);
```

Kodowanie UTF-8 jest popularne i sprawdza się w przypadku polskich znaków, ale spróbuj pobawić się np. US_ASCII lub ISO_8859_1. Jeżeli w wymienionych stałych nie ma kodowania, które nas interesuje, możemy wykorzystać metodę:

```
Charset.availableCharsets().keySet().stream().forEach(System.out::println);

// A następnie wykorzystać
Charset.forName("windows-1250");
```

W praktyce warto jest znać podstawowe rodzaje kodowań np. UTF-8 lub UTF-16, natomiast zabawa tym jest potrzebna najczęściej wtedy gdy wystąpi problem ze znakami diaktrycznymi.

ASCII i Unicode

Omówmy sobie (tak dla przypomnienia) jak działa w Javie typ char.

```
char elementB = 'B';
System.out.println(elementB);
```

Wydrukuje nam B, oczywiście.

Ale możemy użyć też liczby, która reprezentuje dany kod w **ASCII** lub **Unicode**:

```
char elementB = 66;
System.out.println(elementB);
```

Liczba 66 w tablicy znaków ASCII reprezentuje właśnie znak B. Warto wiedzieć, że w tablicy ASCII /Unicode litery wielkie i małe mają przypisane inne liczby, np. małe **b** jest reprezentowane przez liczbę

98.

Procesor pracuje na liczbach, ale my chcemy używać też liter i innych znaków. Jak to zrobić? Właśnie za pomocą kodowania, czyli przypisywania jakiemuś znakowi liczby, która go reprezentuje. Tak powstało ASCII a później Unicode. Unicode jest rozszerzoną wersją ASCII, ASCII bowiem nie zawiera liter i znaków z innych alfabetów. Jeśli chcemy uzyskać taki znak, np małe polskie ł, to musimy posłużyć się liczbą z tablicy Unicode (czyli większym numerem niż daje ASCII). Java i typ char, oczywiście, sobie z tym poradzą. Typ char operuje na tablicy Unicode, zostało to rozwiązane w ten sposób, aby typ ten mógł przechować więcej znaków niż tylko te z tablicy ASCII.

```
char character = 322;
System.out.println(character);
```

zwrócone zostanie ł.

A tutaj znajdziesz linki do tablicy znaków ASCII oraz Unicode.

Unicode vs UTF-8

Jaka jest zatem różnica między Unicode a UTF-8? Nie jest to jedno i to samo.

Unicode jest standardem, który definiuje w jaki sposób można mapować znaki na numery (numery te są inaczej zwane punktami kodowymi).

UTF-8 jest jednym ze sposobów zamiany punktów kodowych na formę zrozumiałą dla komputera, czyli na bity. Można to rozumieć jak algorytm, który pozwala przekonwertować wspomniane punkty kodowe na sekwencję bitów lub odwrotnie - sekwencję bitów na punkty kodowe. Oprócz UTF-8 istnieje również wiele różnych kodowań.

W pokazywanych przykładach kodu, znaki są zamieniane na tablice byte[], stąd widzimy reprezentację w postaci bajtów, a nie bitów. Dodajmy tutaj też, że czytając o kodowaniu w internecie raz natkniesz się na informacje, że kodowanie znaków to sposób zamiany znaków na bajty, a raz, że na bity. Pamiętając, że 1 bajt to 8 bitów, używane jest to wymiennie.

Wróćmy do Streamów

Wracając do Streamów. Powiedzieliśmy, że druga różnica między Reader/Writer, a InputStream /OutputStream to kodowanie plików. Stosując Reader/Writer możemy określić jakie kodowanie znaków nas interesuje, przez co metody będą je stosowały automatycznie. W przypadku InputStream/OutputStream nie mamy takiej możliwości.

Widzimy już, że Streamy znakowe mogą być stosowane do operowania na plikach tekstowych, Streamy bajtowe również mogą być używane do operowania na plikach tekstowych, ale oprócz tego mogą być używane do operowania na obrazkach. Manipulacja obrazkami przy wykorzystaniu Streama tekstowego wydaje się lekko bez sensu ☺.

Jak możemy sobie podzielić Streamy na kategorie?

Streamy będą miały w nazwach takie słowa kluczowe jak Input albo Output, Reader albo Writer. Na tej podstawie możemy wnioskować, czy służą do zapisu czy do odczytu. Jednocześnie Streamy można

podzielić na takie będące na niskim albo wysokim poziomie. Streamy na niskim poziomie, to takie, które przykładowo czytają bajt po bajcie, podczas gdy te na wysokim poziomie odczytują lub zapisują wiele bajtów jednocześnie. Te na wysokim poziomie mogą mieć w nazwie np. `Buffered`. Streamy na wysokim poziomie mogą przyjmować inne Streamy jako swoje punkty wejściowe.

W końcu przykład kodu:

```
try (ObjectInputStream objectStream = new ObjectInputStream(
        new BufferedInputStream(
            new FileInputStream("someFile.txt")))) {
    // reszta kodu
}
```

Pomimo, że ten zapis może się wydawać dziwny, to mamy tutaj Stream niskiego poziomu `FileInputStream`, który odczytuje plik `someFile.txt` bajt po bajcie. Następnie ten Stream jest opakowany Streamem `BufferedInputStream`, który buforuje odczyt, czyli pozwala na wczytywanie kilku bajtów jednocześnie w celu poprawy wydajności. Całość jest owinięta w `ObjectInputStream`, który pozwoli nam odczytać zawartość pliku jako obiekt.

Zapis jak wyżej jest normalny w tym podejściu, nie jest to jakiś rocket-science.

Trzeba również pamiętać, że nie można zestawiać ze sobą `Readerów` i `Writerów` oraz `Input` i `Output` Streamów, czyli:

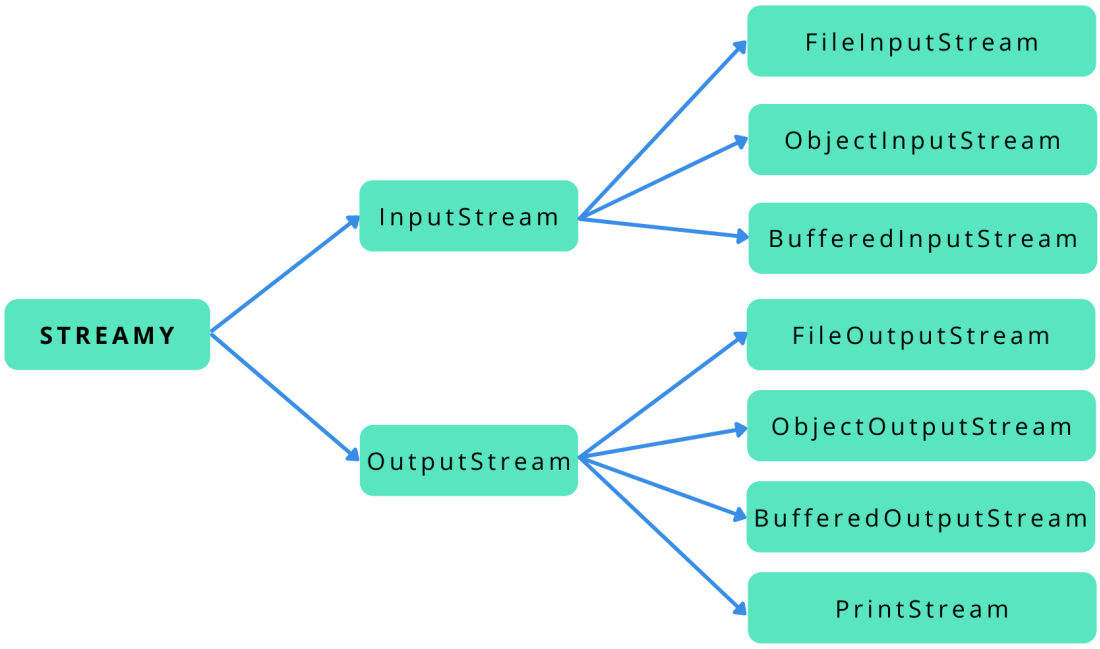
```
// Żaden przykład się nie kompiluje
new BufferedInputStream(new FileWriter("customFile.txt"));
new BufferedWriter(new FileInputStream("customFile.txt"));
new ObjectInputStream(new BufferedOutputStream(new FileOutputStream("customFile.txt")));
```

Poniżej umieszczamy tabelę Streamów, których można używać do operacji opisywanych wcześniej. Nie ucz się jej na pamięć.

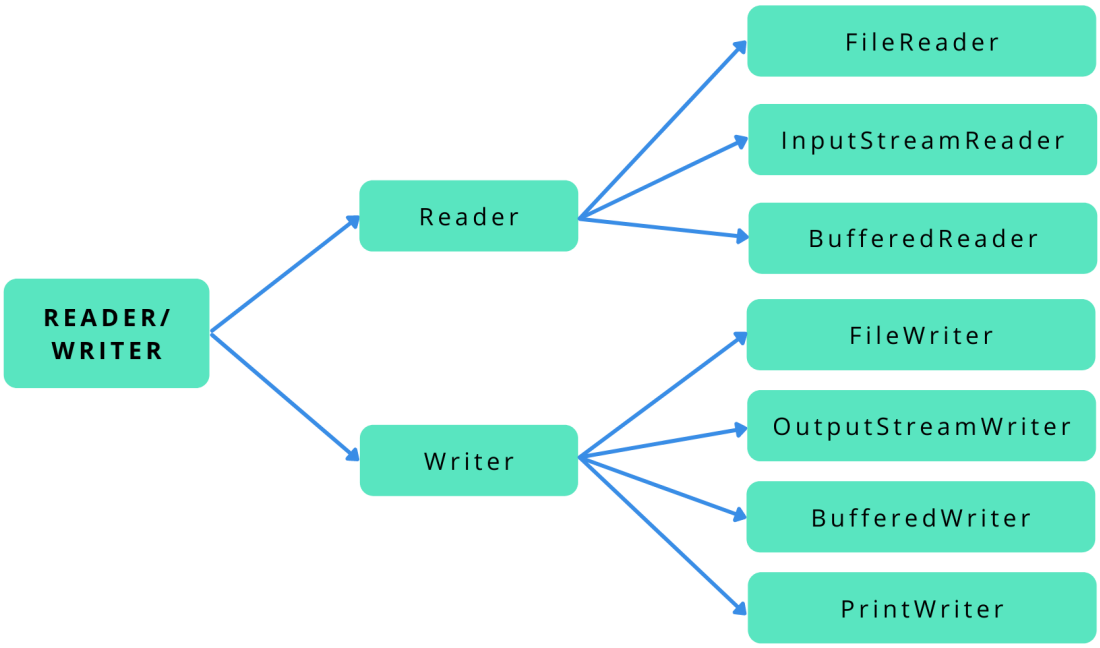
Nazwa klasy	Opis	Poziom
Reader	Klasa abstrakcyjna, z której dziedziczą klasy typu Reader	-
FileReader	Klasa odczytująca pojedyncze znaki z pliku	Niski
InputStreamReader	Klasa odczytująca znaki z podanego InputStreama	Wysoki
BufferedReader	Klasa odczytująca znaki z podanego Readera grupując je w celu poprawy wydajności	Wysoki
Writer	Klasa abstrakcyjna, z której dziedziczą klasy typu Writer	-
FileWriter	Zapisuje dane do pliku jako znaki	Niski
OutputStreamWriter	Klasa zapisująca znaki do podanego OutputStreama	Wysoki
BufferedWriter	Klasa zapisująca znaki do podanego Writera grupując je w celu poprawy wydajności	Wysoki

Nazwa klasy	Opis	Poziom
PrintWriter	Klasa ułatwiająca formatowanie drukowanych treści	Wysoki
InputStream	Klasa abstrakcyjna, z której dziedziczą klasy typu InputStream	-
FileInputStream	Klasa odczytująca informacje z pliku w postaci bajtów	Niski
BufferedInputStream	Klasa grupująca odczyt z InputStreama w celu poprawy wydajności	Wysoki
ObjectInputStream	Klasa będąca w stanie odczytać obiekty Javowe z podanego InputStreama	Wysoki
OutputStream	Klasa abstrakcyjna, z której dziedziczą klasy typu OutputStream	-
FileOutputStream	Zapisuje dane do pliku w postaci bajtów	Niski
BufferedOutputStream	Klasa grupująca zapis do OutputStreama w celu poprawy wydajności	Wysoki
ObjectOutputStream	Klasa będąca w stanie zapisać obiekty Javowe do podanego OutputStreama	Wysoki
PrintStream	Klasa ułatwiająca formatowanie drukowanych treści	Wysoki

A gdyby spróbować to samo przedstawić na grafice to wyglądałoby to tak:



Obraz 1. Hierarchia IO Streamów



Obraz 2. Hierarchia Reader/Writer

Jak żyć z tymi Streamami?

Teraz przyda nam się konstrukcja `try-with-resources`, która była pokazana wcześniej. Stream jest uważany za resource, czyli może być automatycznie zamknięty w bloku `try-with-resources`. Dzieje się tak bo klasy `InputStream`, `OutputStream`, `Reader` i `Writer` implementują interfejs `Closeable`, który implementuje interfejs `AutoCloseable`.

Otwieranie i zamykanie

Na tej podstawie wiemy już, że Stream możemy otworzyć, czyli zarezerwować odpowiednie zasoby, aby móc operować na plikach, zapisywać do nich, lub z nich czytać.

Spuszczanie wody

Kolejnym ciekawym zjawiskiem jest to, że dane, które zapisujemy do Streamów niekoniecznie muszą od razu znaleźć się w pliku, do którego piszemy. Informacje te mogą zostać przetrzymane dłużej w pamięci operacyjnej i dopiero po pewnym czasie zapisane do pliku. Ma to takie konsekwencje, że jeżeli w tym oknie czasowym nasza aplikacja niespodziewanie zakończy działanie, dane te mogą nie zostać zapisane do pliku. Aby takiej sytuacji zapobiec, możemy wykonywać metodę `flush()`, jak spłuczka, możemy ręcznie wypchnąć dane do pliku. Ma to jednak swoją wadę, a mianowicie konsumuje więcej czasu, gdyż operacja faktycznego zapisu do pliku jest kosztowna czasowo. Dlatego powinniśmy jej używać z głową, nie co chwila, tylko przykładowo w odstępach czasu. Jednocześnie należy dodać, że metoda `close()`, która jest wywoływana w bloku `try-with-resources` (pamiętasz, że interfejs `AutoCloseable` definiował taką metodę) również wywołuje `flush()`.

Pomijanie

Mamy również możliwość pomijania znaków w Streamie i służy do tego metoda `skip()`. Nigdy jej chyba nie użyłem w praktyce...

W następnej części notatek będzie trochę praktyki (w końcu)...