Metody programowania

Wprowadzenie do platformy Java

Dr inż. Andrzej Grosser

Częstochowa, 2013

Spis treści

1. Biblioteka wejścia-wyjścia

Rozdział opisuje podstawy biblioteki wejścia-wyjścia. Jest to jedynie zarys możliwości tej biblioteki (zawiera wiele klas). Do napisania tego fragmentu tekstu skorzystano z książek "Core Java"[2],"Thinking in Java"[1], "Learning Java"[3] i dokumentacji Javy.

1.1. Strumienie

Wejście-wyjście Javy jest zdefiniowane w kategoriach strumieni. Strumień jest abstrakcją danych płynących ze źródła aż do ujścia. Takie podejście pozwala ukryć szczegóły związane z dostępem do rzeczywistych urządzeń. Innymi słowy strumienie pozwalają na utworzenie zunifikowanych interfejsów dostępu i odczytu danych, dzięki czemu wygodnie można czytać/zapisywać niezależnie od tego czy dane pochodzą z plików na dysku czy soketów.

1.2. Hierarchia klas

Hierarchia klas wejścia-wyjścia w Javie jest bardzo rozbudowana. Klasy są zgrupowane w pakiety `java.io` i `java.nio` (tzw. nowe wejście-wyjście wprowadzone w Java 1.4). Główne klasy to od których wyprowadza się podstawowe operacje to:

- `InputStream` - klasa abstrakcyjna wyprowadzająca podstawowy interfejs do odczytu bajtów,
- `OutputStream` - klasa abstrakcyjna wyprowadzająca podstawowy interfejs do zapisu bajtów,
- `Reader` - klasa abstrakcyjna wprowadzająca podstawowy intefejs do odczytu znaków (w tym Unicode),
- `Writer` klasa abstrakcyjna wprowadzająca podstawowy intefejs do zapisu znaków (w tym Unicode),
- `File` - klasa wprowadzająca operacje na plikach i katalogach.

Najważniejszymi operacjami, które dostarczają główne klasy strumieni są operacje odczytu dla strumieni czytających (metoda `read()`) i zapisu dla strumieni zapisujących (metoda `write()`). Metody te zwracają/wysyłają sekwencje bajtów/znaków. Dostarczają również kilku innych metod, z których najważniejszą jest `close()` (powoduje zamknięcie strumienia). Klasy zapisujące dostarczają metody `flush()` pozwalającej opróżnić strumień.

1.3. Zapis i odczyt plików binarnych

Na podstawowym poziomie Java udostępnia opisane poniżej metody pozwalające na zapis i odczyt danych (odpowiednio dla klas dziedziczących po InputStream i OutputStream).

Pierwszą z metod jest metoda `read()`, która pozwala na odczyt jednego bajtu, zwraca odczytany bajt lub wartość -1 (gdy natrafi na koniec danych). Są również dostępne przeciążone wersje tej metody pozwalającej na odczyt tablicy bajtów.

Kolejną z metod jest `write()`, która pozwala na zapis jednego bajtu. Dostępne są wersje tej metody pozwalające na zapis tablicy bajtów.

Następna metoda `available()` zwraca ile bajtów jest dostępnych do odczytu w danej chwili. Można jej użyć w następujący sposób:

```
1 int dostepne = plik.available();
2 if(dostepne > 0) {
3     byte[] dane = new byte[dostepne];
4           plik.read(dane);
5 }
```

Przedstawiony powyżej kod zawsze będzie działał dla plików dyskowych.

Ostatnią z opisywanych tutaj metod jest `close()`. Metoda close() powinna być wywoływana po wykorzystaniu do określonych przez programistę celów strumienia. To zalecenie jest to spowodowane względami wydajnościowymi - close zwalnia zasoby systemowe udostępnione strumieniowi. Ponadto zamknięcie strumienia powoduje zapis danych, jakie ewentualnie mogły być przechowywane w dodatkowych buforach (bez zamknięcia strumienia niektóre dane mogłyby nie zostać zapisane).

1.4. Opakowania strumieni

Czytanie sekwencji bajtów i znaków nie jest wygodne i może powodować powstawanie wielu błędów. Dlatego w Javie powstały klasy, których zadaniem jest dodawanie nowych wygodnych funkcji. Przy czym, niektóre z tych klas dostarczają funkcji pozwalających otrzymywać dane z różnych lokalizacji. Inne klasy strumieni dostarczają mechanizmów pozwalających na czytanie wysokopoziomowych danych (np. liczb rzeczywistych). Możliwe jest jednak połączenie korzyści z obydwu rodzajów rozszerzeń, jest to realizowane przez klasy filtry, które opakowuję one odpowiedni strumień - opakowany strumień jest argumentem konstruktora klasy filtru. Na przykład:

```
1 DataInputStream in
2     = new DataInputStream(new FileInputStream("dane.dat"));
3 double x = in.readDouble();
```

W podanym powyżej przykładzie obiekt klasy DataInputStream nie jest w stanie czytać danych z plików, natomiast obiekt klasy FileInputStream dostarcza jedynie metod pozwalających na odczyt bajtów. Obiekt klasy w operacjach odczytu korzysta z obiektu klasy FileInputStream, który dostarcza mu do tworzenia wartości typów liczbowych surowych danych w postaci bajtów z pliku. Obiekt klasy DataInputStream nie musi wiedzieć, skąd rzeczywiście pochodzą otrzymywane bajty, wystarczy, że potrafi ich użyć.

Przykładem klasy filtra jest również `BufferedInputStream`, która dostarcza dodatkowej warstwy, która pozwala buforować wejście. Możliwe jest użycie tej klasy na przykład w sposób następujący (połączenie filtrów w łańcuch - buforowany dostęp do pliku na dysku):

```
1 DataInputStream in =
2     new DataInputStream(new BufferedInputStream
3             (new FileInputStream("dane.dat")));
```

1.5. Pliki o dostępie swobodnym

Klasa `RandomAccessFile` umożliwia zapis i odczyt danych w dowolnym miejscu pliku. Ta klasa umożliwia potraktowanie pliku binarnego jako jednej wielkiej tablicy bajtów. Miejsce, w którym obiekt tej klasy zapisuje lub odczytuje dane jest nazywane wskaźnikiem pliku lub indeksem, ustalenie tego miejsca jest możliwe z wykorzystaniem metody

`seek()`. W sytuacji, gdy będzie próba zapisu lub odczytu poza określonym rozmiarem pliku zostanie wyrzucony wyjątek.

Jako przykład wykorzystania klasy RandomAccessFile można podać następujący kod:

```
//...
//Drugi argument ("w") konstruktora otworzy plik do zapisu -
//Próba odczytu spowoduje powstanie wyjątku.
try(RandomAccessFile plik = new RandomAccessFile("dane.dat","w")
    ) {
    plik.seek(ROZMIAR_REKORDU * numer);
    plik.writeInt(liczba);
    plik.writeDouble(ocena);
}
catch(IOException e) {
    //Obsługa wyjątku
}
```

Powyżej pokazano w jaki sposób otworzyć tego rodzaju plik i zapisać w nim liczbę całkowitą i rzeczywistą. Założono, że plik zawiera na przemian liczby całkowite i rzeczywiste - posiada więc narzuconą strukturę wewnętrzną, dzięki której można przeskakiwać pomiędzy kolejnymi rekordami (można w ten sposób utworzyć prostą bazę danych).

1.6. Zapis i odczyt plików tekstowych

Zanim zostaną omówione sposoby zapisu i odczytu danych należy zwrócić uwagę na jeden aspekt tego procesu - w trakcie odczytu i zapisu danych ustawiane jest kodowanie znaków. Domyślne kodowanie jest pobierane z ustawień systemowych. Jeżeli istnieje potrzeba ustalenia innego sposobu kodowania, należy podać je jako argument dla konstruktora:

```
PrintWriter out = new PrintWriter("plik.txt", "UTF8");
```

Dla powyższego przykładu ustawiono kodowanie utf8 dla zapisywanych danych tekstowych (niezależnie od ustawień systemowych).

1.6.1. Zapis plików tekstowych

Do zapisu danych tekstowych należy użyć klasy `PrintWriter`. Klasa ta dostarcza metod pozwalających na wydrukowanie łańcuchów znaków i liczb w formacie tekstowych (jest jednym z wielu rodzajów opakowania na strumień).

Otwarcie pliku do zapisu jest realizowane instrukcją:

```
1 PrintWriter wrt = new PrintWriter("plik.txt");
```

co jest równoważne:

```
1 PrintWriter wrt = new PrintWriter(new FileWriter("plik.txt"));
```

Konstruktor klasy PrintWriter może przyjąć dodatkową wartość typu logicznego. w ten sposób można ustalić czy ma być wykonywane automatyczne opróżnianie strumienia po wykonaniu metody println (działanie podobne do manipulatora endl w C++).

Zapis danych jest wykonywany za pomocą metod print(), println() i printf(). Są one przeładowane dla liczb, znaków, wartości logicznych, łańcuchów znaków i obiektów. Na przykład:

```
1 wrt.print("Adamski");
2 wrt.print(' ');
3 wrt.print(12.0);
```

Metody print klasy `PrintWriter` nie wywołują wyjątku `IOException`. Błędy trzeba jawnie sprawdzić za pomocą metody `checkError()`.

```
1 wrt.println(dlugiLancuchZnakow);
2 if(wrt.checkError())
3     System.err.println("Zapis nie powiódł się");
```

1.6.2. Odczyt plików tekstowych

Jednym ze sposobów odczytu plików tekstowych jest użycie klasy `BufferedReader`

```
1 BufferedReader in =
2         new BufferedReader(
3                             new InputStreamReader(new
                                  FileInputStream("d.txt")));
```

Dane są odczytywane z pliku za pomocą metody `readline()` - czytana jest linijka tekstu. Odczyt jest realizowany aż do momentu, gdy metoda zwróci wartość null. Typowa interakcja wygląda następująco:

```
1 String line;
2 while((line = in.readline()) != null) {
3     System.out.print(line);
4 }
```

Do odczytu danych z plików tekstowych można użyć też klasy `Scanner`. Umożliwia ona odczyt danych typów podstawowych i łańcuchów znaków przy użyciu wyrażeń regularnych.

Obiekt klasy Scanner rozdziela wejścia na tokeny, granice podziałów są ustalone z wykorzystaniem wzorca separatora, którym domyślnie są białe znaki (może on być ustalany z wykorzystaniem metody `useDelimiter()`). Wynikowe tokeny mogą być konwertowane do wartości różnych typów przy użyciu różnego rodzaju metod, których nazwy zaczynają się od prefiksu next (jest i samo next zwracające łańcuch znaków).

```
1 Scanner in = new Scanner(new File("liczby"));
2 while (in.hasNextLong()) {
3     long liczba = in.nextLong();
4 }
```

Obiekt skanera nie musi być tworzony z wykorzystaniem pliku, można użyć strumienia wejściowego, ścieżki (obiekty `Path`) czy nawet łańcucha znaków:

```
1 String input = "1␣2␣3␣4";
2 Scanner s = new Scanner(input);
```

1.7. Serializacja

Java wspiera mechanizm serializacji, który umożliwia zapis obiektu do strumienia i odczyt obiektu ze strumienia.

Klasa dostarczająca możliwość serializacji musi implementować interfejs `Serializable`. Nie wymaga on implementacji żadnych metod (znakuje jedynie obiekt).

Do zapisu obiektów do strumienia należy otworzyć obiekt klasy `ObjectOutputStream`. Do odczytu obiektów służy `ObjectInputStream`. Obiekty są odczytywane za pomocą metody `readObject()` (zwraca Object, więc konieczne jest rzutowanie do właściwego typu) a zapisywane za pomocą metody `writeObject()`. Na przykład:

```
1 class Osoba implements Serializable {
2     String imie;
3     String nazwisko;
4     public Osoba(String i, String n) {
5         imie = i; nazwisko = n;
6     }
7 }
```

```
8  //...
9  Osoba o = new Osoba("Jan", "Babacki");
10 //Otwarcie strumienia do zapisu
11 ObjectOutputStream out = new ObjectOutputStream(new
       FileOutputStream("osoby.dat"));
12 //Zapis obiektu
13 out.writeObject(o);
14 //Zamknięcie strumenia
15 out.close();
16 //Otwarcie strumienia do odczytu
17 ObjectInputStream in = new ObjectInputStream(new
       FileOutputStream("osoby.dat"));
18 //Odczyt obiektu - konieczne rzutowanie
19 Osoba o2 = (Osoba)in.readObject();
20 //Zamknięcie strumienia
21 in.close();
```

1.8. Praca z plikami systemowymi

Do pracy z plikami systemowymi można użyć klas `File` (ścieżki i operacje plikowe) i wygodniejszego tandemu `Files` (operacje na plikach) - `Path` (ścieżka do plików). Klasy Files i Path zostały wprowadzone w Javie SE 7.

Poniżej podano w kolejnych punktach sposoby wykorzystania klas Path i Files do różnych operacji na plikach i katalogach. Są to krótkie fragmenty kodu sygnalizujące możliwości tych klas, dokładne informacje można poznać z dokumentacji.

1.8.1. Tworzenie, niszczenie plików i katalogów

1. Tworzenie katalogu:

```
1 Path path = Paths.get("katalog");
2 Files.createDirectory(path);
```

2. Tworzenie pliku

```
1 Path path = Paths.get("plik.txt");
2 Files.createFile(path);
```

3. Niszczenie pliku

```
1 Path path = Paths.get("plik.txt");
2 Files.delete(path);
3 //lub
4 bool success = Files.deleteIfExists(path);
```

Metoda delete() powoduje powstanie wyjątku, jeżeli kasowany plik nie istnieje.

1.8.2. Kopiowanie i przenoszenie plików

1. Kopiowanie plików

```
Path src = Paths.get("c:\\plik.txt");
Path dest = Paths.get("c:\\katalog\\plik2.txt");
//Kopiowanie kończy się niepowodzeniem, gdy plik
//docelowy istnieje
Files.copy(src,dest);
//Wymuszenie nadpisywania plików
Files.copy(src,dest,
   StandardCopyOption.REPLACE_EXISTING);
```

2. Przenoszenie plików

```
Path src = Paths.get("c:\\plik.txt");
Path dest = Paths.get("c:\\katalog\\plik2.txt");
//Przesunięcie z nadpisaniem i utrzymaniem atrybutów pliku
Files.move(src,dest,StandardCopyOption.REPLACE_EXISTING,
   StandardCopyOption.COPY_ATTRIBUTES);
```

1.8.3. Przechodzenie przez elementy katalogu

Do przechodzenia poprzez wszystkie składowe katalogu użyteczna jest klasa biblioteczna `DirectoryStream`:

```
try(DirectoryStream<Path> elems = Files.newDirectoryStream(
   directory)
{
   for(Path elem : elems) {
      //Operacje na elementach katalogu
   }
}
```

Literatura

[1] B. Eckel. *Thinking in Java*. Helion, 2006.

[2] C. S. Horstmann and G. Cornell. *Core Java Volume I–Fundamentals*. Prentice Hall, 2011.

[3] P. Niemeyer and D. Leuck. *Learning Java*. O'Reilly Media, 2013.