Programowanie w języku Java - System wejścia-wyjścia (Java I/O System)

mgr inż. Maciej Lasota <m.lasota@tu.kielce.pl>

Version 1.0, 13-05-2017

Spis treści

Zajęcia **laboratoryjne nr 9** z programowania w języku Java - jedne zajęcia **1,5h**.

System wejścia-wyjścia w Javie

Ważnym i bardzo często wykorzystywanym obszarem zastosowań platformy Java jest programowanie aplikacji wykorzystujących dane odczytywane **bezpośrednio od użytkownik**, z **plików** lub z **zasobów sieciowych**. Java dostarcza dwóch podstawowych pakietów, służących do przeprowadzania operacji **wejścia-wyjścia**:

- Pakiet - `java.io.*`
- Pakiet - `java.nio.*`

Pakiet `java.io` jako pakiet *podstawowy* zawiera przede wszystkim klasy, które pozwalają operować na **strumieniach danych**. Klasy reprezentujące strumienie *(inaczej klasy strumieniowe)* są podstawowym środkiem programowania operacji wejścia-wyjścia w Javie.

Pakiet `java.nio` *(Java new input-output)* wprowadzono dodatkowe środki wejścia-wyjścia, takie jak **kanały**, **bufory** i **selektory** . Mimo nazwy *(new input-output)* środki te nie zastępują klas strumieniowych. Służą przede wszystkim do zapewnienia wysokiej *efektywności* i *elastyczności* programów, które w bardzo dużym stopniu obciążone są operacjami wejścia-wyjścia. W szczególności dotyczy to **serwerów**, które muszą równolegle obsługiwać ogromną liczbę połączeń sieciowych.

Oprócz tego Java dostarcza klasy reprezentujące inne od strumieni obiekty operacji **wejścia-wyjścia**. Do klas tych należy np. klasa `File` z pakietu `java.io` - opisująca pliki i katalogi, a także w pakiecie `java.net` - klasy reprezentujące obiekty *"sieciowe"*, takie jak **URL** czy **gniazdo** *(socket)*, mogące stanowić źródło lub odbiornik danych w sieci (w szczególności w Internecie).

Klasy strumieniowe, operacje na strumieniach

Na wstępie należy zdefiniować pojęcie strumienia danych. **Strumień danych** jest pojęciem *abstrakcyjnym*, *logicznym* oznaczającym ciąg danych, właśnie *"strumień"*, do którego dane mogą być **dodawane** i/lub z którego dane mogą być **pobierane**.

Przy czym:

- strumień związany jest ze źródłem lub odbiornikiem danych,
- źródło lub odbiornik mogą być dowolne: plik, pamięć, URL, gniazdo, potok ...,
- strumień służy do zapisywania-odczytywania informacji - dowolnych danych,
- program:
  - kojarzy strumień z zewnętrznym źródłem/odbiornikiem,
  - otwiera strumień,

    - dodaje lub pobiera dane ze strumienia,
    - zamyka strumień.
- przy czytaniu lub zapisie danych z/do strumienia mogą być wykonywane dodatkowe operacje (np. buforowanie, kodowanie-dekodowanie, kompresja-dekompresja),
- w Javie dostarczono klas, reprezentujących strumienie. Hierarchia tych klas pozwala na programowanie w sposób abstrahujący od konkretnych źródeł i odbiorników.

Na strumieniach możemy wykonywać dwie podstawowe operacje:

- **odczytywanie danych**,
- **zapisywanie danych**.

Z tego punktu widzenia możemy mówić o strumieniach *wejściowych* i *wyjściowych*. I odpowiednio do tego – Java wprowadza dwie rozłączne hierarchie klas strumieniowych:

- **klasy strumieni wejściowych**,
- **klasy strumieni wyjściowych**.

Tabela 1: Hierarchie klas strumieniowych.

	Wejście	**Wyjście**
Strumienie bajtowe	InputStream	OutputStream
Strumienie znakowe	Reader	Writer

Są to **klasy abstrakcyjne**, zatem bezpośrednio nie można tworzyć obiektów tych klas.

Przy przetwarzaniu tekstów należy korzystać ze **strumieni znakowych** ze względu na to, iż w trakcie *czytania/pisania* wykonywane są odpowiednie operacje **kodowania/dekodowania** ze względu na stronę kodową właściwą dla źródła/odbiornika.

[Hierarchia klas InputStream] | *input.gif*

Rysunek 1: Hierarchia klas InputStream.

[Hierarchia klas OutputStream] | *output.gif*

Rysunek 2: Hierarchia klas OutputStream.

[Hierarchia klas Reader] | *reader.gif*

Rysunek 3: Hierarchia klas Reader.

[Hierarchia klas Writer] | *writer.gif*

Rysunek 4: Hierarchia klas Writer.

Klasy strumieniowe – przedmiotowe (implementacje)

Źródło/odbiornik	Strumienie znakowe	Strumienie bajtowe
Pamięć	CharArrayReader CharArrayWriter	ByteArrayInputStream ByteArrayOutputStream
	StringReader StringWriter	StringBufferInputStream
Potok	PipedReader PipedWriter	PipedInputStream PipedOutputStream
Plik	FileReader FileWriter	FileInputStream FileOutputStream

Użycie klas przedmiotowych nie jest jedynym sposobem **związania logicznego** strumienia z fizycznym *źródłem* lub *odbiornikiem*. Inne klasy (spoza pakietu `java.io`, np. klasy sieciowe) mogą dostarczać metod, które zwracają jako wynik referencję do **abstrakcyjnego** strumienia związanego z konkretnym źródłem odbiornikiem *(np. plikiem w Sieci)*.

Klasy strumieniowe – przetwarzające (implementacje)

Rodzaj przetwarzania	Strumienie znakowe	Strumienie bajtowe
Buforowanie	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Filtrowanie	FilterReader FilterWriter	FilterInputStream FilterOutputStream
Konwersja: bajty-znaki	InputStreamReader OutputStreamWriter	
Konkatenacja		SequenceInputStream
Serializacja obiektów		ObjectInputStream ObjectOutputStream
Konwersje danych		DataInputStream DataOutputStream
Zliczanie wierszy	LineNumberReader	LineNumberInputStream
Podglądanie	PushbackReader	PushbackInputStream
Drukowanie	PrintWriter	PrintStream

Przykład 1. Przykład użycia klas FileInputStream, FileOutputStream

```
import java.io.*;
public class CopyFile {

   public static void main(String args[]) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;

      try {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Przykład 2. Przykład użycia klas FileReader, FileWriter

```
import java.io.*;
public class CopyFile {

   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Przykład 3. Przykład użycia klas InputStream, OutputStream

```
import java.io.*;
public class FileStreamTest {

   public static void main(String args[]) {

      try {
         byte bWrite [] = {11,21,3,40,5};
         OutputStream os = new FileOutputStream("test.txt");
         for(int x = 0; x < bWrite.length ; x++) {
            os.write( bWrite[x] );   // writes the bytes
         }
         os.close();

         InputStream is = new FileInputStream("test.txt");
         int size = is.available();

         for(int i = 0; i < size; i++) {
            System.out.print((char)is.read() + "  ");
         }
         is.close();
      }catch(IOException e) {
         System.out.print("Exception");
      }
   }
}
```

Buforowanie

Buforowanie ogranicza liczbę fizycznych odwołań do urządzeń zewnętrznych, dzięki temu że fizyczny odczyt lub zapis dotyczy całych porcji danych, gromadzonych w **buforze** *(wydzielonym obszarze pamięci)*. Jedno fizyczne odwołanie wczytuje dane ze strumienia do bufora lub zapisuje zawartość bufora do strumienia.

Przykład 4. Przykład użycia klasy BufferedReader

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.InputStream;
import java.io.InputStreamReader;

public class BufferedReaderDemo {

   public static void main(String[] args) throws Exception {
      String thisLine = null;

      try {

         // open input stream test.txt for reading purpose.
         BufferedReader br = new BufferedReader("c:/test.txt");

         while ((thisLine = br.readLine()) != null) {
            System.out.println(thisLine);
         }
      } catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

Przykład 5. Przykład użycia klasy BufferedWriter

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.StringWriter;

public class BufferedWriterDemo {

   public static void main(String[] args) throws IOException {
      StringWriter sw = null;
      BufferedWriter bw = null;

      String str = "Hello World!";

      try {

         // create string writer
         sw = new StringWriter();

         //create buffered writer
         bw = new BufferedWriter(sw);

         // writing string to writer
         bw.write(str);

         // forces out the characters to string writer
         bw.flush();

         // string buffer is created
         StringBuffer sb = sw.getBuffer();

         //prints the string
         System.out.println(sb);

      } catch(IOException e) {

         // if I/O error occurs
         e.printStackTrace();
      } finally {

         // releases any system resources associated with the stream
         if(sw!=null)
            sw.close();
         if(bw!=null)
            bw.close();
      }
   }
}
```

Serializacja obiektów

Obiekty tworzone przez program rezydują w **pamięci operacyjnej**, w przestrzeni adresowej procesu. Są zatem *nietrwałe*, bo kiedy program kończy działanie wszystko co znajduje się w jego przestrzeni adresowej ulega *wyczyszczeniu* i nie może być odtworzone. **Serializacja** *(szeregowanie)* pozwala na utrwalaniu obiektów. W Javie polega ona na zapisywaniu obiektów do **strumienia**.

Podstawowe zastosowania serializacji:

- komunikacja pomiędzy obiektami/aplikacjami poprzez gniazdka (sockets),
- zachowanie obiektu (jego stanu i właściwości) do późniejszego odtworzenia i wykorzystania przez tę samą lub inną aplikację.

Do zapisywania/odczytywania obiektów służą klasy `ObjectOutputStream` oraz `ObjectInputStream`, które należą do strumieniowych klas przetwarzających.

Do strumieni mogą być zapisywane tylko **serializowalne obiekty**. Obiekt jest serializowalny jeśli jego klasa implementuje interfejs `Serializable`. Prawie wszystkie klasy standardowych pakietów Javy implementują ten interfejs.

Operacje na plikach i katalogach

Zarówno **pliki** jak i **katalogi** w języku Java reprezentowane są przez klasę `java.io.File`. Klasy `File` używamy aby sprawdzić, czy interesujący nas plik lub katalog istnieje, do tworzenia nowych plików oraz katalogów a także aby sprawdzić, czy dana instancja klasy `File` reprezentuje plik czy katalog. Klasa ta umożliwia także takie operacje jak kasowanie oraz zmianę nazwy, natomiast nie służy ona do *odczytu* czy *zapisu* z/do pliku.

Przykład 6. Przykład odczytu danych z pliku

```
public void readFile(String filePath) throws IOException {
  FileReader fileReader = new FileReader(filePath);
  BufferedReader bufferedReader = new BufferedReader(fileReader);

  try {
    String textLine = bufferedReader.readLine();
    do {
      System.out.println(textLine);

      textLine = bufferedReader.readLine();
    } while (textLine != null);
  } finally {
    bufferedReader.close();
  }
}
```

Przykład 7. Przykład zapisu danych do pliku

```
public void writeFile(String filePath, String[] textLines)
    throws IOException {

  FileWriter fileWriter = new FileWriter(filePath);
  BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

  try {
    for (String line : textLines) {
      bufferedWriter.write(line);
      bufferedWriter.newLine();
    }
  } finally {
    bufferedWriter.close();
  }
}
```

Pliki o dostępie swobodnym

Klasa `RandomAccessFile` definiuje pliki o dostępie swobodnym, które mogą być otwarte w trybie *"czytania"* lub *"czytania i pisania"*. Swobodny dostęp oznacza dostęp do **dowolnego bajtu danych** bez potrzeby **sekwencyjnego przetwarzania** pliku od początku. Pliki o dostępie swobodnym mogą być traktowane jako *ciągi bajtów*. Bieżący bajt do odczytu lub miejsce do zapisu określa specjalny *wskaźnik pozycji* w pliku (filePointer). Pozycję tę możemy zmieniać za pomocą metod `seek()` i `skip()`. Jest także zmieniana przy każdej operacji czytania lub pisania. Do czytania/pisania służy wiele metod `read()` i `write()`, które pozwalają operować na różnych rodzajach danych odczytywanych i zapisywanych do pliku *(np. readDouble, readLine, writeInt itp.)*.

Pliki o dostępie swobodnym **nie są strumieniami**. Klasa `RandomAccessFile` nie należy do hierarchii klas strumieniowych.

Przykład 8. Przykład odczytu/zapisu danych do pliku o dostępie swobodnym

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomAccessFileExample {

    public static void main(String[] args) {
        try {
            String filePath = "source.txt";
            System.out.println(new String(readCharsFromFile(filePath, 1, 5)));

            writeData(filePath, "Data", 5);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static void writeData(String filePath, String data, int seek) throws
IOException {
        RandomAccessFile file = new RandomAccessFile(filePath, "rw");
        file.seek(seek);
        file.write(data.getBytes());
        file.close();
    }

    private static byte[] readCharsFromFile(String filePath, int seek, int chars)
throws IOException {
        RandomAccessFile file = new RandomAccessFile(filePath, "r");
        file.seek(seek);
        byte[] bytes = new byte[chars];
        file.read(bytes);
        file.close();
        return bytes;
    }

}
```

Bibliografia

- **Bruce Eckels**, *"Thinking in Java. Edycja polska. Wydanie IV"*, wydawnictwo Helion.
- **Cay S. Horstmann, Gary Cornell**, *"Java. Podstawy. Wydanie IX"*, wydawnictwo Helion.
- **Cay S. Horstmann, Gary Cornell**, *"Java. Techniki zaawansowane. Wydanie IX"*, wydawnictwo Helion.
- **Krzysztof Barteczko**, *"Podstawy programowania w języku Java, PJWSTK"*, http://edu.pjwstk.edu.pl/wyklady/ppj/scb/
- **Konrad Kurczyna**, *"Laboratorium Java"*, Politechnika Świętokrzyska w Kielcach.

- **Mariusz Lipiński**, *"Nauka Javy"*, http://www.naukajavy.pl/