Rozdział 7 Strumienie, operacje wejścia-wyjścia

7.1 Strumienie

W celu stworzenia uniwersalnego modelu przepływu danych wykorzystuje się w Javie model strumieni danych. Strumień (*stream*) jest sekwencją danych, zwykle bajtów. Pochodzenie i typ sekwencji danych zależny jest od danego środowiska. Podstawowe typu strumieni to te związane z operacjami wprowadzania danych do programu (operacje wejścia) i z operacjami wyprowadzania danych poza program (operacje wyjścia). W Javie do obsługi operacji wejścia stworzono klasę *InputStream*, natomiast dla obsługi operacji wyjścia stworzono klasę *OutputStream*. Strumień związany jest również z typem obszaru (urządzenia) z/do którego sekwencja danych przepływa oraz typem danych. Podstawowe obszary (urządzenia) to np. pamięć operacyjna, dyski (pliki), sieć, drukarka, ekran, itp. Typy danych jakie mogą być wykorzystywane przy przesyłaniu danych to np. *byte, String, Object*, itp. Zgodnie z dystrybucją JDK 1.0 klasy *OutputStream* oraz *InputStream* reprezentują strumienie jako sekwencje bajtów, czyli elementów typu *byte*. Często jednak zachodzi potrzeba formatowania danych strumienia. Można to wykonać w Javie korzystając z różnych klas formatujących dziedziczących z *OutputStream* i *InputStream*. Dobrym przykładem jest obsługa danych tekstowych wyświetlanych na standardowym urządzeniu wyjścia. Zgodnie z tym co było omawiane wcześniej klasyczną metodą wydruku tekstu w konsoli jest użycie polecenia *System.out.println();* gdzie *out* jest obiektem klasy *PrintStream*, stanowiącej klasę formatującą bajty sekwencji pochodzących z *OutputStream* na tekst. Rozpatrując dalej różnorodność obsługi strumieni w Javie należy wspomnieć o dodatkowych klasach wprowadzonych z wersją JDK 1.1, a mianowicie klasach *Reader* oraz *Writer*. Klasy te stanowią analogię do klas *InputStream* oraz *OutputStream*, niemniej przygotowane są do obsługi danych tekstowych (*String*). W omawianych wcześniej rozdziałach użyto już przykładowej klasy dziedziczącej z klasy *Writer*, a mianowicie klasy *PrintWriter* (zamiast *PrintStream*). Wprowadzenie dodatkowych klas obsługujących sekwencje łańcuchów znaków miało na celu ujednolicenie pracy w środowisku Javy z tekstem zapisywanym kodowanym w Unicodzie (2 bajty na znak). Dodatkowe, oddzielne klasy strumieni to klasa *StreamTokenizer* dzieląca strumień tekstowy na leksemy oraz klasa *RandomAccessFile* obsługująca pliki zawierające rekordy o znanych rozmiarach, tak że można dowolnie poruszać się w obrębie rekordów i je modyfikować. Ważnym zagadnieniem związanym ze strumieniami jest możliwość zapisu obiektu jako sekwencji bajtów i przesłanie go do programu (metody) działającej zdalnie. Efekt ten jest uzyskiwany poprzez zastosowanie mechanizmu serializacji i wykorzystania klas *ObjectInputStream* oraz *ObjectOutputStream*.

Wszystkie omawiane klasy obsługujące różne typy strumieni są zdefiniowane w pakiecie java.io.
Wykorzystanie strumieni jest powszechne w tworzeniu programów tak więc warto bliżej zapoznać się z podstawowymi klasami je obsługującymi.

7.2 Standardowe obsługa wejścia-wyjścia - klasy InputStream oraz OutputStream

7.2.1 Obsługa wejścia – klasa InputStream

Klasa ta jest klasą abstrakcyjną i zawiera podstawowe metody umożliwiające odczytywanie i kontrolę bajtów ze strumienia. Ponieważ jest to klasa abstrakcyjna nie tworzy się dynamicznie obiektu tej klasy, lecz można go uzyskać poprzez odwołanie się do standardowego wejścia zainicjowanego zawsze w polu *in* klasy *System*, czyli *System.in*. Inne możliwości uzyskania obiektu klasy *InputStream* to wywołanie metod zwracających referencję do obiektu tego typu np.: metoda *getInputStream*() zdefiniowana w klasie *Socket*. Jedyną metodą abstrakcyjną (czyniącą z tej klasy klasę abstrakcyjną) jest metoda *read*() oznaczająca czytanie kolejnego bajtu ze strumienia wejścia. Pozostałe metody umożliwiają:

- odczyt bajtów do zdefiniowanej tablicy:

```
int read(byte b[]);
int read(byte b[], int offset, int length);
```

- pominięcie określonej liczby bajtów w odczycie:

```
long skip(long n);
```

- kontrolę stanu strumienia (czy są dane):

```
int available();
```

- tworzenie markerów:

```
boolean markSupported(); kontrola czy tworzenie markerów jest możliwe
synchronized void mark(int readlimit);
synchronized void reset()
```

- zamknięcie strumienia

```
void close();
```

Prawie wszystkie metody (poza *markSupported*() oraz *mark*()) mogą wygenerować wyjątek, który musi być zadeklarowany lub obsługiwany w kodzie programu. Podstawową klasą wyjątków jest tutaj klasa *IOException*.
Pokazane wyżej metody *read*() blokują dostęp tak długo, jak dane są dostępne lub wystąpi koniec pliku albo wygenerowany zostanie wyjątek.
Klasy dziedziczące z klasy *InputStream* to:

- ByteArrayInputStream - definiuje pole bufora bajtów,
- FileInputStream - umożliwia odczyt pliku (strumień z pliku),
- FilterInputStream - praktycznie kopiuje funkcjonalność InputStream celem jej wykorzystania w klasach dziedziczących z FilterInputStream wprowadzającą dodatkowe narzędzia do obsługi sekwencji bajtów:

BufferedInputStream, CheckedInputStream, DataInputStream, DigestInputStream, InflaterInputStream, LineNumberInputStream, ProgressMonitorInputStream, PushbackInputStream

np. DataInputStream umożliwia programowi odczyt danych zgodnie z podstawowymi formatami zdefiniowanymi w Javie (char, int, long, double, itp.).

- ObjectInputStream - dokonuje rekonstrukcji obiektu z sekwencji bajtów (stworzonej w wyniku serializacji i zapisu metodą writeObject() klasy ObjectOutputStream),
- PipedInputStream - dokonuje przepływu danych do odpowiadającemu obiektowi klasy PipedOutputStream, który jest podawany jako argument konstruktora klasy PipedInputStream,
- SequenceInputStream, dokonuje logicznej koncentracji strumieni w jeden,
- StringBufferInputStream - tworzy strumień z podanego łańcucha znaków (obiektu String) - klasa ta jest oznaczona w JDK 1.2 jako deprecated ponieważ nie dokonuje właściwej konwersji znaków na bajty. Postuluje się jej zastąpienie klasą StringReader.

7.2.2 Obsługa wejścia – klasa OutputStream

W podobny sposób, niemniej dotyczący obsługi wyjścia, definiowane są klasy dziedziczące z klasy *OutputStream*. Klasa ta jest również klasą abstrakcyjną z jedyną abstrakcyjną metodą *write*() zapisująca kolejny bajt do strumienia. Podstawowe metody tej klasy to:

void close() - zamknięcie strumienia,
void flush() - przesuwa buforowane dane do strumienia,
void write(byte[] b, int off, int len),
void write(byte[] b) - zapisują dane z tablicy b do strumienia wyjścia.

Klasy dziedziczące z klasy OutputStream to:

- ByteArrayOutputStream,
- FileOutputStream,
- FilterOutputStream,
- ObjectOutputStream,
- PipedOutputStream.

Klasy te obsługują strumień wyjściowy a ich funkcjonalność jest analogiczna do omawianych wyżej wersji obsługujących wejście.

Przykład 7.1:

```
//Echo.java

import java.io.*;
```

```
public class Echo{

	public static void main(String args[]){
		byte b[] = new byte[100];
		try{
			System.in.read(b);
			System.out.write(b);
			System.out.flush();
		} catch (IOException ioe){
			System.out.println("Błąd wejścia-wyjścia");
		}

	}

}//koniec public class Echo
```

Powyższy program ukazuje proste zastosowanie strumieni. Początkowo wykorzystywany jest istniejący strumień wejścia *System.in* w celu wczytania danych ze standardowego urządzenia wejścia (klawiatura). Dane są wczytywane aż wciśnięty zostanie klawisz *Enter* (koniec danych). Wczytane znaki są zapisywane do bufora b, który to jest następnie wykorzystywany do pobrania danych celem wysłania ich do strumienia wyjścia. Przyjętym w powyższym przykładzie strumień wyjścia to *System.out*. Warto zauważyć, że wykorzystano metody *write*() oraz *flush*() do zapisu danych. W efekcie działania programu otrzymujemy echo (powtórzenie) napisu wprowadzonego z klawiatury.

Kolejny przykład ukazuje możliwość dostępu do pliku.

Przykład 7.2:

```
//ZapiszPlany.java:

import java.io.*;

class Plany implements Serializable{

	private int liczbaLegionow;
	private int liczbaDzial;
	private String haslo;

	public void ustaw(int lL, int lD, String h){
		this.liczbaLegionow=lL;
		this.liczbaDzial=lD;
		this.haslo=h;
	}
	public void wyswietl(){
		System.out.println("Liczba legionów = "+liczbaLegionow);
		System.out.println("Liczba dział = "+liczbaDzial);
		System.out.println("Hasło dostępu = "+haslo);
	}
}// koniec class Plany

class Nadawca extends Thread{
```

```
	private String plikDanych;
	ZapiszPlany zp;
	Nadawca(String s, ZapiszPlany zp){
		this.plikDanych=s;
		this.zp=zp;
		setName("Nadawca");
	}
	public void run(){
		byte b[] = new byte[100];
		try{
			System.out.println("Podaj hasło");
			System.in.read(b);
			String s= new String(b);
			System.out.println("Zapisuję do pliku");
			Plany p = new Plany();
			p.ustaw(1,2,s);
			ObjectOutputStream o = new ObjectOutputStream(new
FileOutputStream(plikDanych));
			o.writeObject(p);
			o.flush();
			o.close();
		} catch (IOException ioe){
			System.out.println("Błąd wejścia-wyjścia");
		}
		zp.ustaw();
	}

}// koniec class Nadawca

class Odbiorca extends Thread{
	private String plikDanych;
	ZapiszPlany zp;
	Odbiorca(String s, ZapiszPlany zp){
		this.plikDanych=s;
		this.zp=zp;
		setName("Odbiorca");

	}
	public void run(){
		while( (!(this.isInterrupted())) && (zp.pobierz()) ){

		}
		try{
			System.out.println("Odczyt");
			ObjectInputStream i = new ObjectInputStream(new FileInputStream(plikDanych));
			Plany p = (Plany) i.readObject();
			p.wyswietl();
			i.close();
		} catch (Exception e){
			System.out.println("Błąd wejścia-wyjścia lub brak klasy Plany");
		}


	}
```

```
}// koniec class Odbiorca


public class ZapiszPlany{
        static boolean czekaj=true;
        synchronized void ustaw(){
                czekaj=false;
        }
        synchronized boolean pobierz(){
                return czekaj;
        }
        public static void main (String args[]){
                ZapiszPlany z = new ZapiszPlany();
                Nadawca n = new Nadawca("plany.txt",z);
                Odbiorca o = new Odbiorca("plany.txt",z);
                o.start();
                n.start();

        }

}//koniec public class ZapiszPlany
```

Powyższy program demonstruje zastosowanie obsługi strumieni dostępu do plików oraz wykorzystanie serializacji obiektów. Stworzono w kodzie cztery klasy. Pierwsza zawiera definicje zbioru pól (dane) oraz metod dostępu do nich. Klasa implementuje interfejs *Serializable* w celu umożliwienia zapisu obiektu do strumienia. Kolejne dwie klasy opisują zachowanie wątków generującego zapis danych i odczytującego zapis danych. W klasie Nadawca zawarto kod wczytujący zestaw znaków w klawiatury, który jest przypisywany do pola obiektu klasy Plany. Następnie tworzony jest strumień dostępu do pliku o podanej nazwie sformatowany do przesyłania obiektów. Zapis do bufora i przesłanie do strumienia odbywa się poprzez metody *writeObject*() oraz *flush*(). Po zakończeniu procesu zapisu danych do pliku powiadamiany jest drugi wątek, opisany w klasie Odbiorca. Zapisany obiekt jest odczytywany poprzez metodę *readObject*() a następnie jest wykonywana jedna z metod obiektu wyświetlająca stan danych. Jak widać przesłanie przez strumień obiektów jest ciekawym rozwiązaniem i umożliwia tworzenie rozproszonych aplikacji.

7.2.3 Obsługa plików

Dostęp do plików zaprezentowany wcześniej wykorzystywał klasy *FileInputStream* i *FileOutputStream*. Konstruktory tych klas umożliwiają inicjacje strumienia poprzez podanie jako argumentu albo nazwy pliku poprzez obiekt typu *String* lub poprzez podanie nazwy logicznej reprezentowanej przez obiekt klasy *File*. Klasa *File* opisuje abstrakcyjną reprezentację ścieżek dostępu do plików i katalogów. Ścieżka dostępu do pliku może być sklasyfikowana ze względu na jej zasięg lub ze względu na środowisko dla którego jest zdefiniowana. W pierwszym przypadku dzieli się ścieżki dostępu na absolutne i relatywne. Absolutne to te, które podają adres do pliku względem głównego korzenia systemu plików danego środowiska. Relatywne to te, które adresują plik względem katalogu bieżącego. Druga klasyfikacja rozróżnia ścieżki dostępu pod względem środowiska dla którego jest ona zdefiniowana, co w praktyce dzieli ścieżki dostępu na te zdefiniowane dla systemów opartych na UNIX i

na te zdefiniowane dla systemów opartych na MS *Windows* (własność systemu o nazwie file.separator). Przykłady:

a.) absolutna ścieżka dostępu:
UNIX: /utl/software/java/projekty
MS Windows: c:\utl\softare\java\projekty

b.) relatywna ścieżka dostępu:
UNIX: java/projekty
MS Windows: java\projekty.

Tworząc obiekt klasy *File* dokonywana jest konwersja łańcucha znaków na abstrakcyjną ścieżkę dostępu do pliku (abstrakcyjna ścieżka dostępu do pliku jest tworzona według określonych reguł podanych w dokumentacji API). Metody klasy *File* umożliwiają liczne kontrolę podanej ścieżki i plików (np. *isFile(), isDirectory(), isHidden, canRead(),* itp.) oraz dokonywania konwersji (np*. getPath(), getParent(), getName(), toURL(),* itp.) i wykonywania prostych operacji (*list() mkdir(),* itp.).

Uwaga! Należy pamiętać, że zapis tekstowy ścieżki dostępu dla środowiska MS Windows musi zawierać podwójny separator, gdyż pojedynczy znak umieszczony w inicjacji łańcucha znaków oznacz początek kodu ucieczki, np. „c:\\java\\kurs\\wyklad\\np".

Przykład 7.3:

```
// PobierzDane.java:

import java.io.*;

public class PobierzDane{

        public static void main(String args[]){
                File f = new File("DANE1");
                if (f.mkdir()) {
                        File g = new File (".");
                        String s[] = g.list();
                        for (int i =0; i<s.length; i++){
                                System.out.println(s[i]);
                        }
                } else {
                        System.out.println("Błąd operacji I/O");
                }

        }


}//koniec public class PobierzDane
```

Program PobierzDane ukazuje ciekawą własność obiektu *File*. Otóż stworzenie obiektu tej klasy nie oznacza otwarcia strumienia czy stworzenia uchwytu do pliku. Obiekt klasy *File* może więc być stworzony praktycznie dla dowolnej nazwy ścieżki. W prezentowanym przykładzie utworzono katalog o nazwie „DANE1”, a następnie

dokonano wydruku plików i katalogów zawartych pod aktualnym adresem ścieżki (pod tym, z którego wywołano program **java**).

Pracę z plikami o swobodnym dostępie ułatwia zastosowanie innej klasy obsługującej operacje wejścia-wyjścia, a mianowicie klasy *RandomAccessFile*. Zdefiniowana jest w klasie obsługa plików zawierających rekordy o znanych rozmiarach, tak że można dowolnie poruszać się w obrębie rekordów i je modyfikować. Dane w pliku są interpretowane jako dane w macierzy do której dostęp jest możliwy poprzez odpowiednie ustawienie głowicy czytającej czy zapisującej dane. Zdefiniowano w tej klasie metody przesuwania głowicy (*getFilePointer*(), *seek*()) oraz szereg metod czytania i zapisu różnych typów danych.

7.3 Obsługa strumieni tekstu

W związku z problemem wynikającym z konwersji znaków Javy (Unicode) na bajty i odwrotnie występujących we wczesnych (JDK 1.0) realizacjach klas obsługi strumieni począwszy od wersji JDK1.1 wprowadzono dodatkowe klasy *Reader* i *Writer*. Obie abstrakcyjne klasy są analogicznie skonstruowane (dziedziczenie z klasy *Object* i deklaracja metod) jak klasy *InputStream* oraz *OutputStream*. Dziedziczące z nich klasy umożliwiają prostą i formatowaną obsługę sekwencji tekstu:

Reader:
- BufferedReader – buforuje otrzymywany tekst,
  - LineNumberReader – przechowuje dodatkowo informacje o numerze linii
- CharArrayReader – wprowadza bufor znaków do odczytu,
- FilterReader – klasa abstrakcyjna formatowania danych tekstowych,
  - PushbackReader – przygotowuje dane odesłania do strumienia,
- InputStreamReader – czyta bajty zamieniające je na tekst według podanego systemu kodowania znaków,
  - FileReader – odczyt danych tekstowych z pliku dla domyślnego systemu kodowania znaków, poprzez podanie ścieżki zależnej systemowo (String) lub abstrakcyjnej (File)
- PipedReader – obsługa potoku (związanie z klasą odczytującą),
- StringReader – obsługa strumienia pochodzącego od obiektu klasy String.

Konstrukcja klasy Writer jest analogiczna do Reader, z tym, że definiowany jest zapis zamiast odczytu. Podobnie wygląda struktura klas dziedziczących z Writer:

Writer:
- BufferedWriter,
- CharArrayWriter,
- FilterWriter,
- OutputStreamWriter,
  - FileWriter
- PipedWriter,
- PrintWriter, - formatowanie danych do postaci tekstu (analogiczna do PrintStream)

StringWriter

Odczyt danych odbywa się poprzez zastosowanie metod *read*() lub *readLine*() natomiast zapis danych do strumienia poprzez wykorzystanie metod write(). Zastosowanie klas dziedziczących z *Reader* i *Writer* ma dwie zasadnicze zalety: właściwa konwersja bajtów na znaki w Unicodzie i odwrotnie oraz możliwość zdefiniowania systemu kodowania znaków. W celu zobrazowania sposobu korzystania z tych klas warto zapoznać się z następującym przykładem:

Przykład 7.4:

```
//Czytaj.java:

import java.io.*;

/*Plik „plik.txt” powinien zawierać polskie znaki wprowadzone w kodzie Cp1250: "ąśęółżźćń"; */

public class Czytaj{

        public static void main(String args[]){
          String s;

          char zn[] = new char[9];
          try{
         InputStreamReader r = new InputStreamReader((new FileInputStream("plik.txt")), "Cp1250");
         r.read(zn,0,zn.length);
        s = new String(zn);
        System.out.println(s);
                OutputStreamWriter o = new OutputStreamWriter((new FileOutputStream("plik1.txt")),"Cp852");
        o.write(s,0,s.length());
        o.flush();
        } catch (Exception e){}

        }

}// koniec public class Czytaj
```

Prezentowany program tworzy dwa strumienie: jeden wejścia czytający plik tekstowy zapisany według strony tekstowej Cp1250 (Windows PL) i drugi wyjścia zapisujący nowy plik wyjściowy tekstem według strony kodowej Cp852 (DOS PL). Oczywiście nazwy plików jak i nazwy stron kodowych można zmieniać dla potrzeb ćwiczeń i w ten sposób dokonywać konwersji plików z np. Cp1250 na ISO8859-2. W celu weryfikacji działania powyższego programu należy otworzyć plik o nazwie „plik.txt” w edytorze obsługującym Cp1250 a plik o nazwie „plik1.txt” w edytorze obsługującym Cp852. W przypadku gdy nie ma konieczności zmiany systemu kodowania znaków warto wykorzystywać klasy *FileReader* oraz *FileWriter* zamiast *InputStreamReader* i *OutputStreamWriter*. Dla poprawienia efektywności pracy istotne jest również buforowanie czytanych danych co w rezultacie prowadzi do następującego wywołania obiektu:

```
BufferedReader br = new BufferedReader(new FileReader(”plik.txt”));
```

7.4 Dzielenie strumienia – klasa StreamTokenizer

Na zakończenie omawiania klas związanych z obsługą strumieni warto zapoznać się z klasą *StreamTokenizer*, dzieląca strumień tekstowy na leksemy. Klasa ta daje więc swoistą funkcjonalność wykrywanie elementów strumienia i umieszczania ich w tablicy. Wskazując znak oddzielający leksemy można dokonać przeformatowania przesłanego tekstu (np. podzielić ścieżkę dostępu, dokonać detekcji liczb w tekście, itp.). Pobranie leksemu z tablicy odbywa się poprzez wywołanie metody *nextToken*(). Poniższy przykład ilustruje stosowanie klasy *StreamTokenizer*:

Przykład 7.5:

```
//FormatujStrumien.java:

import java.io.*;

public class FormatujStrumien{

	public static void main(String args[]){
		System.out.println("Podaj tekst zawierający znak : .");
		 Reader r = new BufferedReader(new InputStreamReader(System.in));
		StreamTokenizer st = new StreamTokenizer(r);
		st.ordinaryChar('.');
		st.ordinaryChar('-');
		st.ordinaryChar('A');
		try{
			while (st.nextToken() != StreamTokenizer.TT_EOF){
				if (st.ttype==StreamTokenizer.TT_WORD){
					System.out.println(new String(st.sval));
				}

			}
		}catch (IOException ioe){
			System.out.println("Błąd operacji I/O");
		}

	}


}//koniec public class FormatujStrumien
```

W powyższym przykładzie stworzono obiekt klasy *Reader* na podstawie standardowego strumienia wejścia, który jest następnie wykorzystany przy inicjowaniu obiektu klasy *StreamTokenizer*. Ponieważ w zbiorze domyślnych znaków dzielących strumień nie ma znaków '.' oraz '-' dodano te znaki za pomocą metody *ordinaryChar*(). Dodatkowo ustawiono znak 'A' jako znak dzielący strumień. Następnie w bloku instrukcji warunkowej uruchomiona jest pętla działająca tak długo aż nie wystąpi koniec pliku (aż nie wciśnięty zostanie kod CTRL-Z a następnie

Enter). W pętli pobierany jest kolejny wczytywany element, sprawdzany jest jego typ i jeśli jest to słowo (tekst) to drukowany jest na ekranie tekst będący wartością pola *sval* obiektu klasy *StreamTokenizer*. Obsługa programu polega na wprowadzaniu tekstu ze znakami go dzielącymi i kończeniu linii wciskając *Enter*. W ten sposób linia po linii można analizować wprowadzany tekst. Koniec pracy programu jest wymuszany sekwencją końca pliku CTRL-Z i *Enter*.

Warto zauważyć, że istnieje klasa *StringTokenizer* o podobnym działaniu, której argumentem nie jest jednak strumień a obiekt klasy *String*.

7.5 Strumienie poza java.io

W JDK istnieją jeszcze inne strumienie zdefiniowane poza pakietem *java.io*. Przykładowo w pakiecie *java.util.zip* zdefiniowano szereg klas strumieni obsługujących kompresję w formie ZIP i GZIP. Podstawowe klasy strumieni tam przechowywane to:

CheckedInputStream
CheckedOutputStream
DeflaterOutputStream
GZIPInputStream
GZIPOutputStream
InflaterInputStream
ZipInputStream
ZipOutputStream

Przykładowo w celu dokonania kompresji pliku metodą GZIP można zastosować następujący kod:

Przykład 7.6:

```
//Kompresja.java:

import java.io.*;
import java.util.zip.*;

public class Kompresja{

        public static void main(String args[]){
          String s;

        byte b[] = new byte[100];
        for (int i=0; i<100; i++){
                b[i]=(byte) (i/10);
        }

         try{
                FileOutputStream o = new FileOutputStream("plik2.txt");
                o.write(b);
                o.flush();
                o.close();
```

```
                FileOutputStream fos = new FileOutputStream("plik2.gz");
                GZIPOutputStream z = new GZIPOutputStream(new BufferedOutputStream(fos));
                z.write(b,0,b.length);
                z.close();

            } catch (Exception e){}

 }

}// koniec public class Kompresja
```

W prezentowanym kodzie tworzona jest tablica bajtów wypełniana kolejnymi wartościami od 1 do 10. Tablica ta jest następnie przesyłana do strumieni wyjściowych raz bezpośrednio do pliku, drugi raz poprzez kompresję metodą GZIP. W wyniku działania programu uzyskuje się dwa pliki: bez kompresji „plik2.txt" i z kompresją „plik2.gz".

W pakietach standardowych jak i w pakietach będących rozszerzeniem bibliotek Javy można znaleźć jeszcze szereg innych strumieni związanych z przesyłaniem danych (np. w kryptografii czy w obsłudze portów).

Ze względu na liczne potrzeby wykorzystywania portów szeregowych i równoległych komputera Sun opracował pakiet rozszerzenia Javy o nazwie javax.comm. Pakiet ten umożliwia obsługę portów poprzez strumienie. Poniższy przykład ukazuje próbę zapisu do portu szeregowego.

Przykład 7.7:

```
//ZapiszPort.java:

import java.io.*;
import java.util.*;
import javax.comm.*;

public class ZapiszPort {
  static SerialPort port;
  static CommPortIdentifier id;
  static Enumeration info;
  static String dane = "Tu Czerwona Jarzębina - odbiór \n";
  static OutputStream os;

   public static void main(String args[]) {
      info = CommPortIdentifier.getPortIdentifiers();

      while (info.hasMoreElements()) {
         id = (CommPortIdentifier) info.nextElement();
         if (id.getPortType() == CommPortIdentifier.PORT_SERIAL) {
            if (id.getName().equals("COM1")) {
               try {
                  port = (SerialPort) id.open("ZapiszPort", 2000);
               } catch (PortInUseException e) {}
               try {
```

```
                os = port.getOutputStream();
            } catch (IOException ioe) {}
            try {
                port.setSerialPortParams(9600,
                SerialPort.DATABITS_8,
                SerialPort.STOPBITS_1,
                SerialPort.PARITY_NONE);
            } catch (UnsupportedCommOperationException ue) {}
            try {
                os.write(dane.getBytes());
            } catch (IOException e) {}
        }
      }
    }
  }
}// koniec public class ZapiszPort
```