Programowanie obiektowe

Wykład 9: Mechanizmy wejścia/wyjścia

dr inż. Marcin Luckner
mluckner@mini.pw.edu.pl

Wydział Matematyki i Nauk Informacyjnych

Wersja 1.3
4 marca 2021

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Politechnika Warszawska
Unia Europejska Europejski Fundusz Społeczny

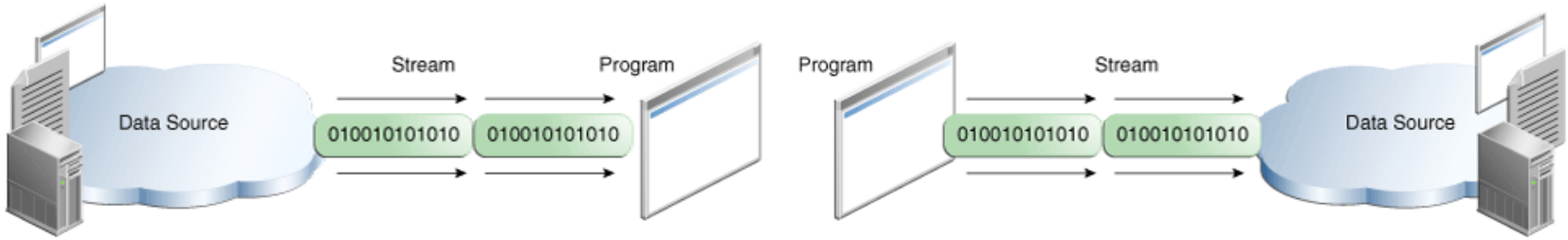
Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Mechanizmy wejścia/wyjścia w Javie

- Pracując z danymi prędzej czy później stajemy przed koniecznością wczytywania danych z plików lub innych zewnętrznych źródeł.
- Równocześnie, w przypadku złożonych analiz, stajemy przed koniecznością trwałego zapisywania wytrenowanych narzędzi analitycznych lub wyników prac.
- Java pozwala przeprowadzić te operacje korzystając z mechanizmu strumieni wejścia i wyjścia.

Kierunek działania strumieni



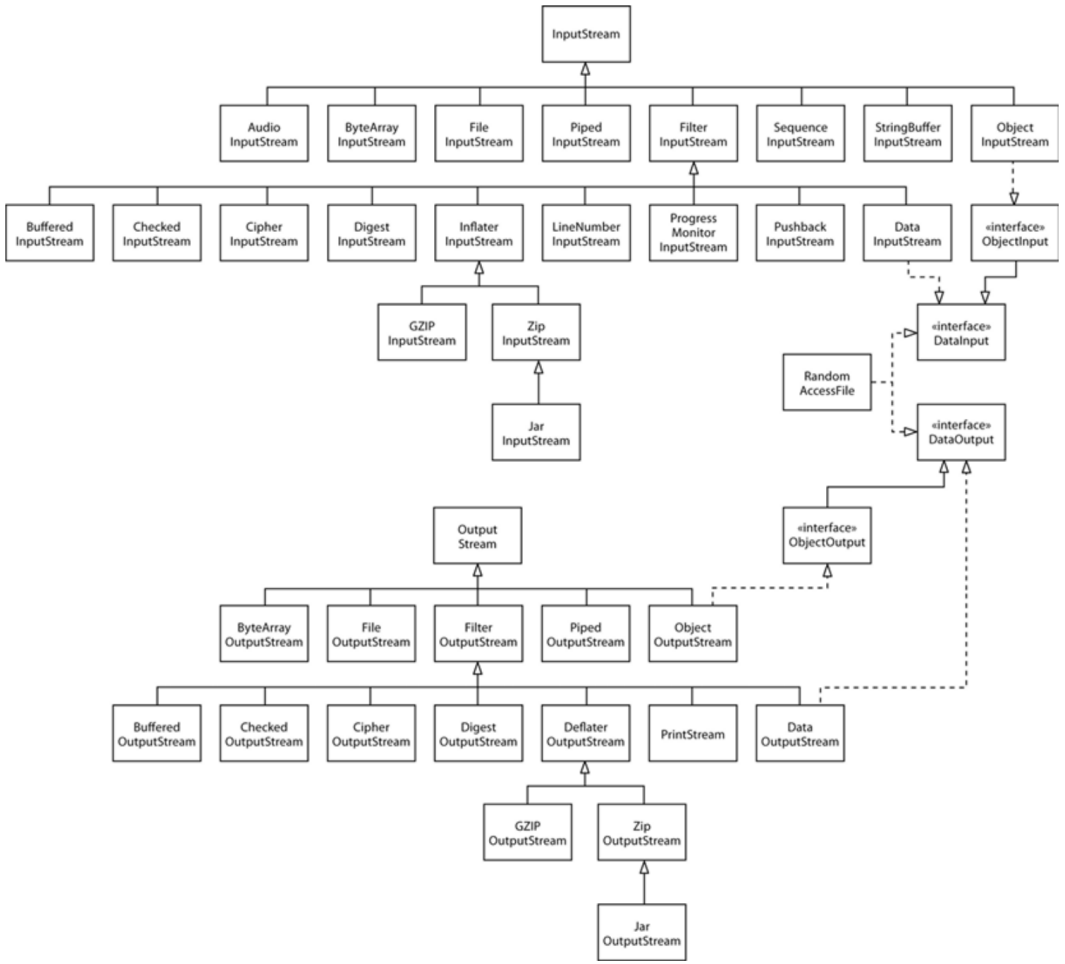
Rysunek 1: Kierunek działania strumieni wejścia i wyjścia [Oracle, 2018]

- Strumienie wejścia wczytują dane z zewnętrznych źródeł do programu.
- Strumienie wyjścia wysyłają dane na zewnątrz.
- Na jednym z końców strumienia zawsze znajduje się program.

Strumienie

- Strumienie pozwalają odczytać lub zapisać sekwencję bajtów.
- Klasą bazową dla strumienia jest
  - `InputStream` dla strumieni wejścia.
  - `OutputStream` dla strumieni wyjścia.
- Klasy bazowe są nadpisywane przez szereg klas dostosowanych do obsługi różnych źródeł i rozszerzających możliwości strumieni.

Hierarchia strumieni



Rysunek 2: Hierarchia strumieni wejścia/wyjscia [Horstmann, 2017]

Przykładowe strumienie

- Strumienie wejścia

  FileInputStream odczyt z pliku.
  GZipInputStream rozpakowywanie w locie.
  DataInputStream odczyt typów prostych.
  BufferedInputSteam odczyt buforowany.
  AudioInputStream odczyt danych audio.

- Strumienie wyjścia

  FileOutputStream zapis do pliku.
  GZipOutputStream pakowanie w locie
  DataOutputStream zapis typów prostych.
  BufferedOutputSteam zapis buforowany.
  PrintStream wyświetlanie danych.

- Zestawy strumieni wejścia i wyjścia nie pokrywają się całkowicie.

Schemat używania strumieni

1. Strumień jest tworzony przez konstruktor.
2. W zależności od kierunku strumienia możemy wywołać metodę `read` lub `write` do odczytu lub zapisu jednego bajtu.
3. Pracę ze strumieniem kończymy wywołując metodę `close`.

Metoda `read`

```
1  int read();
2  int read(byte[] b);
3  int read(byte[] b, int off, int len);
```

- Zwraca typ `int`, ale czyta pojedynczy bajt i zwraca liczbę z przedziału $[0, 255]$.
- Jeżeli osiągnie koniec strumienia to zwraca -1
  - dlatego nie zwraca typu `byte`.
- Na czas odczytu danych blokuje wykonywanie wątku. Program kontynuuje działanie gdy:
  - dane zostaną odczytane,
  - osiągnięty zostanie koniec strumienia,
  - wyrzucony zostanie wyjątek.
- Jeżeli bajt nie może zostać odczytany, a nie osiągnięto końca strumienia to wyrzucany jest wyjątek `IOException`.

Metoda `write`

```
1  void write(int n);
2  void write(byte[] b);
3  void write(byte[] b, int off, int len);
```

- Zapisuje młodszy bajt argumentu i ignoruje resztę.
- Na czas zapisu danych blokuje wykonywanie wątku.
- Jeżeli nie można dokonać zapisu wyrzucany jest wyjątek `IOException`.

Metoda `close`

```
1   void close();
```

- Zamyka strumień wejścia.
- Zamknięty strumien nie może być ponownie użyty, należy utworzyć nowy strumień.
- Konieczne jest zamykanie strumieni.
  - Metoda `close` uwalnia zasoby systemu operacyjnego, równoczesne otwarcie zbyt wiele strumieni może naruszyć stabilność systemu.
  - Zamknięcie strumienia wyjścia powoduje opróżnienie bufora używanego przez ten strumień, Jeżeli nie zamkniemy strumienia, ostatni pakiet bajtów może zostać utracony.
    - Można wymusić opróżnienie bufora metodą `flush`.

Odczyt i zapis bajtów

- Chociaż odczyt i zapis do plików opiera się na przedstawionym schemacie to w praktyce nie jest on używany.
  - Zapis nie jest buforowany. Każda operacja na bajcie odwołuje się do funkcji systemu operacyjnego, spowalniając pracę.
  - Java oferuje znacznie wygodniejsze narzędzia do pracy z danymi.
- W praktyce korzystamy ze strumieni dedykowanych do pewnych zadań np. pracy z plikami czy zapis i odczyt typów podstawowych.
- Można je wykorzystać w uniwersalny sposób dzięki możliwości zagnieżdżania strumieni.

Zagnieżdżanie strumieni

- Chcemy zapisać do pliku serię pomiarów będących liczbami rzeczywistymi.
- Możemy zapisać dane do pliku korzystając z klasy `FileOutputStream`.

```
1   FileOutputStream measurmentFile = new
        FileOutputStream("measurment.dat");
```

- Jednakże strumień `FileOutputStream` oferuje tylko zapis metodą `write`.
- Rozwiązaniem jest połączenie kilku strumieni, aby buforować dane i móc bezpośrednio zapisywać liczby rzeczywiste.

```
1   DataOutputStream output =
2      new DataOutputStream(
3         new BufferedOutputStream(
4            new FileOutputStream("measurment.dat")));
5
6   output.writeDouble(3.4);
```

Kolejność zagnieżdżania strumieni

- Strumienie są tak skonstruowane, że możemy je dowolnie zagnieżdżać.
- Powinniśmy kierować się jednak pewną logiką
  - Wewnętrzny strumień opisuje źródło danych.
  - Zewnętrzny strumień dostarcza potrzebny nam interfejs.
  - Wewnątrz znajdują się strumienie modyfikujące strumień (filtry).

Zapis danych

```
DataOutputStream output =
 new DataOutputStream(
  new BufferedOutputStream(
   new FileOutputStream("file.dat")));

output.writeDouble(3.4);
output.close();
```

Odczyt danych

```
DataInputStream input =
 new DataInputStream(
  new BufferedInputStream(
   new FileInputStream("file.dat")));

System.out.println(input.readDouble());
input.close();
```

Strumienie plikowe

- Operacje na plikach są obsługiwane przez strumienie plikowe
  - `FileOutputStream`
  - `FileInputStream`
- Plik na którym operujemy jest określany alternatywnie poprzez
  - Nazwę i ścieżkę (`String`)
  - Obiekt klasy `File`
  - Deskryptor istniejącego pliku

Klasa `File`

- Klasa `File` udostępnia uniwersalne operacje w systemie plików, takie jak: tworzenie katalogów, kasowanie plików, zmiana nazwy, itp. Więcej w [Horstmann, 2017].
- Obiekt jest definiowany przez nazwę i ścieżkę.
- Ze względu na przenośność kodów Javy należy budując ścieżkę używać atrybutu `File.separator` do określania zagnieżdżeń katalogów zamiast używać znaków przypisanych do danego systemu operacyjnego.

Buforowanie strumienia

- Podstawowe strumienie nie są buforowane. Dla każdego bajtu wywoływane są polecenia systemowe, co zwiększa czas zapisu.
- Strumienie buforowane tworzą bufor do gromadzenia danych, a operacje systemowe są wywołane tylko po zapełnieniu/opróżnieniu bufora.
- Buforowany strumień wejścia `BufferedInputStream`
  - Czyta dane z pamięci (bufora)
  - API natywne jest wywoływane tylko gdy bufor jest pusty
- Buforowany strumień wyjścia `BufferedOutputStream`
  - Zapisuje dane do bufora
  - API natywne jest wywoływane tylko gdy bufor jest pełny

Operacje na typach prostych

- Strumienie DataInputStream i DataOutputStream dostarczają interfejsy pozwalające bezpośrednio odczytywać i zapisywać wszystkie typy proste.
- Są to metody postaci `readTyp` i `writeTyp`.

DataInputStream

```
boolean   readBoolean()
byte    readByte()
char    readChar()
double    readDouble()
float   readFloat()
int   readInt()
long    readLong()
short   readShort()
String    readUTF()
```

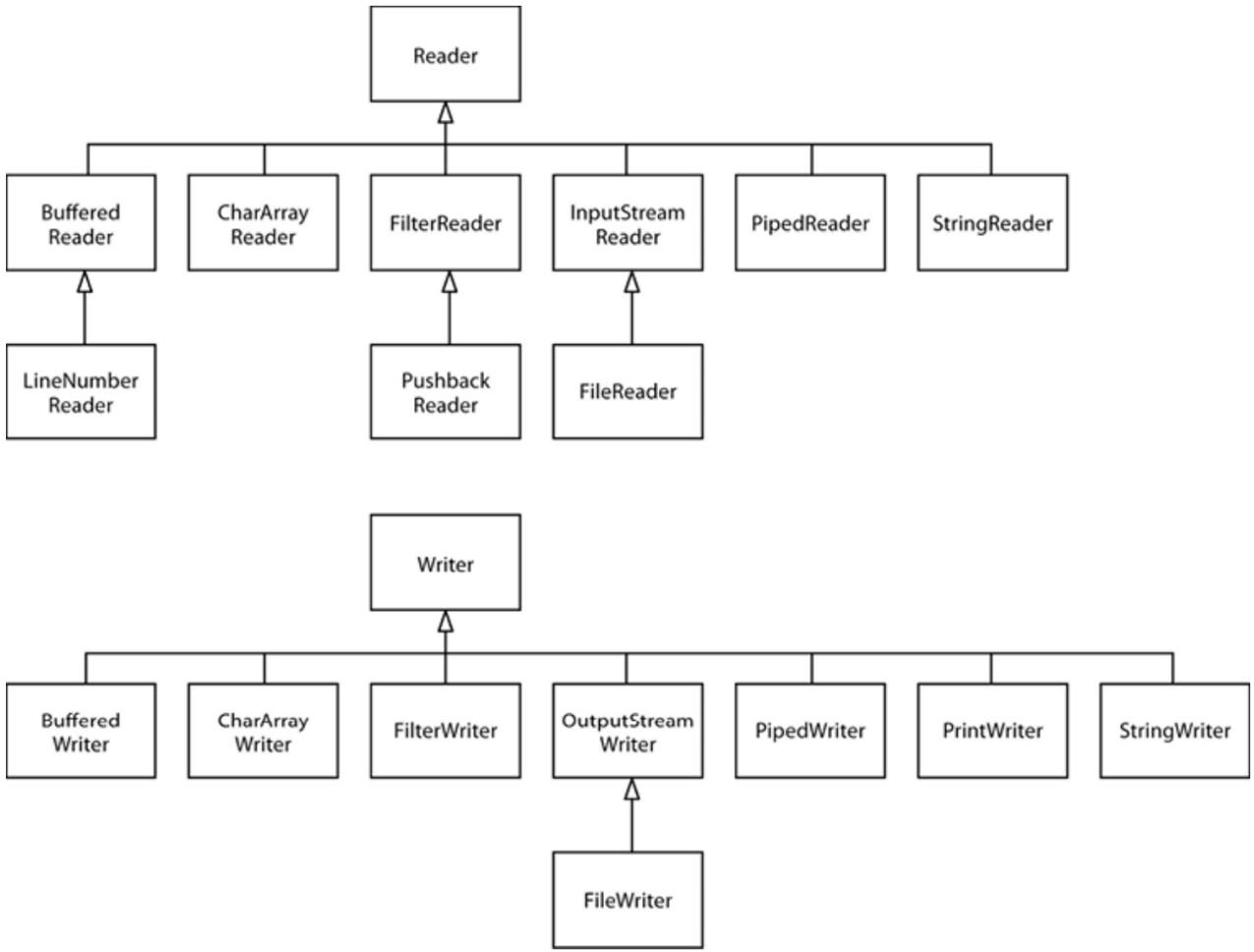
DataOutputStream

```
void    writeBoolean(boolean v)
void    writeByte(int v)
void    writeChar(int v)
void    writeDouble(double v)
void    writeFloat(float v)
void    writeInt(int v)
void    writeLong(long v)
void    writeShort(int v)
void    writeUTF(String str)
```

Strumienie tekstowe

- Przedstawione techniki służą do obsługi danych binarnych, jednakże często mamy do czynienia z danymi w postaci tekstowej.
- O ile zapis binarny jest szybszy i bardziej efektywny, to pozostaje kompletnie nieczytelny dla człowieka.
- Do obsługi danych tekstowych Java używa strumieni tekstowych obsługiwanych przez klasy:
  - `Reader`
  - `Writer`
- Strumienie dokonują konwersji danych reprezentujących znaki za pomocą określonego kodowania na obiekt udostępniający znaki Unicode i *vice versa*.

Hierarchia klas Reader i Writer



Rysunek 3: Hierarchia strumieni do przetwarzania danych tekstowych [Horstmann, 2017]

Przykładowe strumienie tekstowe

- Strumienie wejścia

InputStreamReader konwersja do Unicode.

FileReader odczyt z pliku tekstowego.

StringReader analiza tekstu.

BufferedReader odczyt buforowany.

- Strumienie wyjścia

OutputStreamWriter konwersja z Unicode do innego kodowania.

FileWriter zapis do pliku.

CharArrayWriter zapis do tablicy znaków.

PrintWriter wyświetlanie danych tekstowych.

- Schemat działania strumieni tekstowych jest podobny do zastosowanego w strumieniach binarnych.

Odczyt danych tekstowych

- Możemy czytać dane bezpośrednio z konsoli.

Odczyt z konsoli

```
1  System.out.println("Podaj nazwe pliku:");
2  BufferedReader in = new BufferedReader(
3     new InputStreamReader(System.in));
4  String fileName = in.readLine();
```

- Możemy także odczytywać dane z pliku.

Odczyt z pliku

```
1  BufferedReader in = new BufferedReader(
2     new InputStreamReader(
3        new FileInputStream("data.txt"),
              StandardCharsets.UTF_8));
```

- W przypadku odczytu należy zadeklarować kodowanie znaków.
- W obydwu wypadkach zastosowano klasę `BufferedReader`, aby móc odczytywać całe linie tekstu metodą `readLine`.

Zapis danych tekstowych

- Zapis danych odbywa się przy użyciu klasy `PrintWriter`.
- Klasa potrafi dokonywać zapisu bezpośrednio do pliku

Zapis do pliku

```
1  PrintWriter out = new PrintWriter("data.txt", "UTF-8");
       //rownowazne
2  PrintWriter out = new PrintWriter(
3     new FileOutputStream("data.txt"), "UTF-8");
```

- Klasa jest wykorzystywana do wyświetlania tekstów na konsoli.

Wyświetlanie na konsoli

```
1  System.out.print("Wczytano ");
2  System.out.print(lineNo);
3  System.out.println(" linii tekstu."); // nastepna linia
```

- Znak końca linii jest różny w różnych systemach i jest zwracany metodą `System.lineSeparator`.

Proste przetwarzanie tekstu

- Klasa Scaner pozwala nam na proste przetwarzanie tekstu.

Zapis do pliku

```
1  String input = "1;Robert Lewandowski;10.5;12";
2  Scanner s = new Scanner(input).useDelimiter(";"); //separator kolumn
3  System.out.println(""+s.nextInt()+". "+s.next() +  " zarabia " +
       s.nextDouble()/s.nextInt() + " milionów miesięcznie.");
4  s.close();
```

Wynik

```
1. Robert Lewandowski zarabia 0.875 milionów miesięcznie.
```

- Metoda musi znać strukturę danych i nie jest odporna na zmianę typów danych i występujące w danych błędy.

Proste przetwarzanie strumieni tekstowych

- Klasa `StreamTokenizer` dzieli dowolny strumień tekstowy na tokeny, czyli umowne słowa.
- Kolejne tokeny pozyskujemy metodą `nextToken`.
- Tokeny są rozdzielane definiowanymi znakami białymi.
- Klasa `StreamTokenizer` potrafi rozpoznawać niektóre typy tokenów np. liczby.

Analiza tekstu

```
1  String zadanie = "Ala kupiła na wyprzedaży trampki Converse na
       koturnie za 279.00 złotych i buty New Balance WL574POA za 199
       złotych. Ile razem wydała Ala?";
2  StreamTokenizer st = new StreamTokenizer(new StringReader(zadanie));
3  double sum = 0;
4  try {
5     while(st.nextToken() != StreamTokenizer.TT_EOF) {
6        if(st.ttype == StreamTokenizer.TT_NUMBER) {
7           sum += st.nval;
8        }
9     }
10 } catch (IOException e) {
11    e.printStackTrace();
12 }
13 System.out.println("Ala wydała "+String.format("%6.2f",sum)+ "
       złotych."); //Ala wydała 478.00 złotych.
```

Wyrażenia regularne

- Java pozwala na bardziej zaawansowaną analizę tekstu korzystając z wyrażeń regularnych.
  - Możemy wyszukać elementy opisane wyrażeniem regularnym.

Numer dowodu osobistego

```
[A-Z]{3} \d{6}
```

Poprawne daty ze stycznia

```
[1-2]\d{3}-01-((0[1-9])|([1-2][0-9])|(3[0-1]))
```

  - Możemy zlokalizować je w tekście i dokonać ich edycji korzystając z klas `Pattern` i `Matcher`.

Zastosowanie wyrażeń regularnych

- Klasa `Pattern` określa wyrażenie regularne definiujące wzorzec.
- Klasa `Matcher` pozwala zlokalizować odpowiadające mu fragmenty tekstu, a następnie modyfikować je.

Wyszukiwanie danych z lutego

```
1  String invoices =
       "data;numer;kwota\n2018-01-14;1;12.54\n2018-01-16;2;56.43\n"+
2      "2018-02-01;3;87.5\n2018-02-11;4;17.25\n2018-03-04;5;28.0\n";
3  Pattern p =
       Pattern.compile("[1-2]\\d{3}-02-((0[1-9])|(1[0-9])|(2[0-8])).*");
4  Matcher m = p.matcher(invoices);
5  while(m.find()) {
6      System.out.println(invoices.substring(m.start(), m.end()));
7  }
```

Wynik

```
2018-02-01;3;87.5
2018-02-11;4;17.25
```

Jak utworzyć własny strumień?

- Strumień musi umożliwiać wstawienie pomiędzy dwa inne strumienie
  - Musi nadpisywać odpowiednią klasę (`InputStream`, `OutputStream`, `Writer`, `Reader`)
    - Pozwala na użycie jako argument konstruktora przy tworzeniu innego strumienia
  - Musi mieć konstruktor akceptujący strumień jako argument
    - Pozwala na przekazanie innego strumienia

Przetwarzanie szeregu czasowego

- Mamy obliczyć zmiany poziomu wody na podstawie pomiarów wysokości słupa wody

Przetwarzanie danych

```
1  String data = "5.1;5.2;5.2;5.1;5.3;5.4";
2  Scanner s = new Scanner(new StringReader(data)).useDelimiter(";");
3  double previousHigh = 5;
4  while(s.hasNext()) {
5     double currentHigh = s.nextDouble();
6     System.out.print(String.format("%4.2f",currentHigh-previousHigh)
7     +", ");
8     previousHigh = currentHigh;
9  }
```

Wynik

```
0.10, 0.10, 0.00, -0.10, 0.20, 0.10,
```

Przetwarzanie szeregu czasowego - błędne dane

- Co się stanie, gdy w wyniku błędu aparatury jeden z pomiarów wyniesie `null`?
  - Postać danych wejściowych: data =”5.1;5.2;null;5.1;5.3;5.4”;

Wynik

Exception in thread ”main” java.util.InputMismatchException

- Spróbujemy rozwiązać ten problem tworząc strumień zastępujący braki średnią wyliczoną z ciągu danych.

Klasa NullReplacingReader

Strumień usuwający braki

```
 1  public class NullReplacingReader extends Reader {
 2     private Reader input;
 3     String previousData ="";
 4     static char delimiter =';';
 5
 6     public NullReplacingReader(Reader input) {
 7        super();
 8        this.input=input;
 9     }
10     @Override
11     public int read(char[] cbuf, int off, int len) throws
           IOException {
12        String line = prepareFullDataLine(cbuf, off, len);
13        if(line.length()==0)
14           return previousData.length()==0?-1:0;
15        line = replaceNull(line);
16        line = prepareDataToRead(line,cbuf, off, len);
17        return line.length();
18     }
19     @Override
20     public void close() throws IOException {
21        input.close();
22     }
23  }
```

Funkcje pomocnicze I

- Funkcje pomocnicze obsługują odczyt z bufora i zapis do bufora.

Odczyt tekstu z bufora

```
1  private String prepareFullDataLine(char[] cbuf, int off, int len) {
2     String line;
3     int res = 0;
4     try {
5        res = input.read(cbuf, off, len);
6     } catch (IOException e) {
7        e.printStackTrace();
8     }
9     if (res==len) { //tekst nie mieści się w jednym buforze
10       previousData = previousData +
               String.valueOf(cbuf).substring(0,res);
11    }
12    if(res == -1) { //wczytano juz cały tekst
13       line = previousData;
14    }else { // tekst zmieścił się w jednym buforze
15       line = previousData + String.valueOf(cbuf).substring(0, res);
16    }
17    return line;
18 }
```

Funkcje pomocnicze II

Wpisywanie poprawionego tekstu do bufora

```
1  private String prepareDataToRead(String line, char[] cbuf, int off,
       int len) {
2      if(line.length()>len) { //tekst nie zmieści się w jednym
           buforze, trzeba go podzielić
3          previousData = line.substring(len);
4          line = line.substring(0,len);
5      }
6      for(int i=off;i<line.length(); i++) {
7          cbuf[i] = line.charAt(i);
8      }
9      return line;
10 }
```

Wyliczanie średniej

- Chcąc wyliczyć wartość zastępującą braki musimy wybrać tylko te pomiary, które nie są wartościami `null`.
- Wykorzystamy do tego `StreamTokenizer`.

Wyliczanie średniej

```
1  private Double calculateAverage(String line) {
2     double value = 0;
3     int count = 0;
4     StreamTokenizer st = new StreamTokenizer(new StringReader(line));
5
6     try {
7        while(st.nextToken() != StreamTokenizer.TT_EOF) {
8           if(st.ttype == StreamTokenizer.TT_NUMBER) {
9              count++;
10             value += st.nval;
11          }
12       }
13    } catch (IOException e) {
14       e.printStackTrace();
15    }
16    return value/count;
17 }
```

Eliminacja braków

- Wyliczoną wartość wstawiamy w miejsce wszystkich wystąpień `null`.
- Wykorzystamy do tego klasy `Pattern` i `Matcher`.

Zamiana wartości `null`

```
1  private String replaceNull(String line) {
2     Pattern p = Pattern.compile("null");
3     Matcher m = p.matcher(line);
4     line = m.replaceAll(String.format("%4.2f",
          calculateAverage(line)));
5     return line;
6  }
```

Wykorzystanie klasy `NullReplacingReader`

- Dodajemy klasę `NullReplacingReader` do łańcucha strumieni.

```
1  String data ="5.1;5.2;null;5.1;5.3;5.4";
2  Scanner s = new Scanner(new NullReplacingReader(new
       StringReader(data))).useDelimiter(";");
```

- Strumień zastępuje wartość `null` wyliczoną średnią 5.22.
- Aplikacja zwraca poprawny wynik

Wynik

```
0.10, 0.10, 0.02, -0.12, 0.20, 0.10,
```

- Co istotne zagnieżdżenie strumienia nie zmienia w żaden sposób kodu, który obsługuje zwracane wyniki.

Zapis obiektów

- Do tej pory zapisywaliśmy podstawowe typy danych w formie binarnej lub tekstowej o płaskiej strukturze.
- Z użyciem tych technik można zapisać stan obiektu, ale rodzi to pewne problemy.
  - Gdy obiekt zawiera inne obiekty to tworzy strukturę hierarchiczną, trudną do zapisania w formie pliku płaskiego.
  - Zmiany pól obiektu muszą być odzwierciedlane w metodach do zapisu plików.
- Odpowiadając na te problemy Java wprowadza *serializację*.

Serializacja

- Serializacja zapewnia możliwość zapisu i odczytu dowolnych obiektów implementujących interfejs `Serializable`.

```
1  public class Person implements Serializable
```

- Deklaracja implementacji interfejsu ma tylko znaczenie informacyjne, bo nie posiada on metod, które trzeba nadpisać.
- Umożliwia jednak przekazywanie obiektów jako argumentów do strumieni zapisujących i odczytujących obiekty.

Strumienie obiektowe

- Zapis i odczyt obiektów umożliwiają strumienie obiektowe
  - Strumień `ObjectOutputStream` pozwala zapisywać obiekty metodą `writeObject`

Zapis

```
1  Person batman = new Person("Bruce Wayne",LocalDate.of(1939, 5,
       1));
2  Employee robin = new Employee("Richard Grayson",
       LocalDate.of(1940, 4, 1),batman);
3  ObjectOutputStream out = new ObjectOutputStream(new
       FileOutputStream("batmanFamily.dat"));
4  out.writeObject(robin); //Zapisuje także informacje o
       pracodawcy
```

  - Strumień `ObjectInputStream` pozwala odczytywać obiekty metodą `readObject`

Odczyt

```
1  ObjectInputStream in = new ObjectInputStream(new
       FileInputStream("batmanFamily.dat"))
2  Employee dick = (Employee) in.readObject();
3  //Employee{boss=Person{name='Bruce Wayne',
       birthday=1939-05-01}}
```

Zasady serializacji obiektów

- Pola statyczne i z własnością `transient` są pomijane.
- Obiekty wskazane jako referencje są serializowane rekursywnie.
- W przypadku zawartości nieserializowanej wyrzucony zostanie wyjątek `NotSerializableException`.
- Podklasy serializowanej klasy też są serializowane.
- Podklasa nieserializowanej klasy może być serializowana, jeżeli jej klasa bazowa ma konstruktor bezparametryczny.

Na czym polega serializacja?

- Zapis
  1. Każdy obiekt przekazany do strumienia otrzymuje unikalny numer seryjny.
  2. Jeżeli referencja do obiektu została przekazana po raz pierwszy to obiekt jest zapisywany.
     - Zapisowi podlega identyfikator klasy i stan obiektu.
  3. Jeżeli referencja wskazuje na obiekt o numerze seryjnym, który jest identyczny z numerem już zapisanego obiektu to nie zapisuje obiektu ponownie, a zapisuje tylko informację o ponownym wystąpieniu obiektów.
- Odczyt
  1. Pierwsze wystąpienie zapisu obiektu w strumieniu powoduje utworzenie nowego obiektu i jego inicjalizację danymi ze strumienia. Zostaje zwrócona referencja do obiektu.
  2. Wystąpienie zapisu z numerem seryjnym identycznym z numerem już utworzonego obiektu powoduje zwrócenie referencji do poprzednio utworzonego obiektu.

Wielokrotna serializacja obiektu

- Utwórzmy obiekt `robin`, a następnie zmieńmy jego stan i dokonajmy serializacji

Test serializacji

```
1  Employee robin = new Employee("Richard Grayson", LocalDate.of(1940,
       4, 1),batman);
2  out.writeObject(robin);
3  robin.setName("Jason Todd");
4  robin.setBirthday(LocalDate.of( 1983, 3, 1));
5  out.writeObject(robin);
6
7  System.out.println(((Employee) in.readObject()).whoAmI());
8  System.out.println(((Employee) in.readObject()).whoAmI());
```

- Stan pierwszego wczytanego obiektu:
  - Person{name='Richard Grayson', birthday=1940-04-01}
- Stan drugiego wczytanego obiektu:
  - Person{name='Richard Grayson', birthday=1940-04-01}
- Poprawne działanie uzyskamy tworząc nowy obiekt.

```
1  robin = new Employee("Jason Todd ", LocalDate.of( 1983, 3,
       1),batman);
```

Wersjonowanie

- Co się stanie jeżeli zmienimy definicję klasy dla serializowanego obiektu?.
- Java nada nowej klasie inny identyfikator i zapisy będą różne.
- Możemy wymusić zachowanie typu ustalając wartość pola `static final long serialVersionUID` na wartość charakteryzującą klasę.
- Jak zachowa się serializacja odczytując dane przy istnieniu wielu wersji klasy?
  - Metody
    - Dodawanie i usuwanie metod nie wpływa na serializację.
    - Zmiana zwracanego typu istniejącej metody powoduje brak kompatybilności.
  - Pola
    - Dodane pola nie są serializowane.
    - Usunięte pola są serializowane z wartosciami 0 lub `null`.

Bibliografia

[Horstmann, 2017] Horstmann, C. S. (2017).
Java. Techniki zaawansowane.
Helion.

[Oracle, 2018] Oracle (2018).
I/o streams.