MARCIN LIS

WYDANIE IV

Java

PRAKTYCZNY KURS

Helion

Redaktor prowadzący: Ewelina Burska
Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/pkjav4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Listingi wykorzystane w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/pkjav4.zip

ISBN: 978-83-246-9663-5

Printed in Poland.

Spis treści

Rozdział 6.

System wejścia-wyjścia

Do pisania aplikacji w Javie niezbędna jest znajomość przynajmniej podstaw systemu wejścia-wyjścia. Tej tematyce jest poświęcony ten rozdział. W Javie operacje wejścia-wyjścia są wykonywane za pomocą strumieni. Strumień to abstrakcyjny ciąg danych, który działa — mówiąc w uproszczeniu — w taki sposób, że dane wprowadzone na jednym jego końcu pojawiają się na drugim końcu. Strumienie dzielą się na wejściowe i wyjściowe. Strumienie wyjściowe mają początek w aplikacji i koniec w innym urządzeniu, np. na ekranie czy w pliku, umożliwiają zatem wyprowadzanie danych z programu. Przykładowo standardowy strumień wyjściowy jest reprezentowany przez obiekt `System.out`, a wykonanie metody `println` tego obiektu powoduje, że dane domyślnie zostają wysłane na ekran. Strumienie wejściowe działają odwrotnie: ich początek znajduje się poza aplikacją (może to być np. klawiatura albo plik dyskowy), a koniec w aplikacji, umożliwiają zatem wprowadzanie danych. Co więcej, strumienie umożliwiają też komunikację między obiektami w obrębie jednej aplikacji. W tym rozdziale zostanie jednak omówiona tylko komunikacja aplikacji ze światem zewnętrznym.

Lekcja 31. Standardowe wejście

Opisy podstawowych operacji wyjściowych, czyli wyświetlania informacji na ekranie konsoli, pojawiły się już wielokrotnie; operacje te zostaną dokładniej przedstawione w ramach kolejnej lekcji. Omówienie systemu wejścia-wyjścia rozpoczniemy od wprowadzania danych i standardowego strumienia wejściowego reprezentowanego przez obiekt `System.in`. Tej tematyce zostanie poświęcona cała bieżąca lekcja. Zostaną w niej przedstawione informacje, jak odczytywać dane wprowadzane przez użytkownika z klawiatury.

Standardowy strumień wejściowy

Standardowy strumień wejściowy jest reprezentowany przez obiekt `System.in`, czyli obiekt `in` zawarty w klasie `System`. Jest to obiekt typu `InputStream`, klasy reprezentującej strumień wejściowy. Metody udostępniane przez tę klasę są zebrane w tabeli 6.1. Jak widać, nie jest to imponujący zestaw, niemniej jest to podstawowa klasa operująca na strumieniu wejściowym.

Tabela 6.1. *Metody udostępniane przez klasę InputStream*

Deklaracja metody	Opis
`int available()`	Zwraca przewidywaną liczbę bajtów, które mogą być pobrane ze strumienia przy najbliższym odczycie.
`void close()`	Zamyka strumień i zwalnia związane z nim zasoby.
`void mark(int readlimit)`	Zaznacza bieżącą pozycję w strumieniu.
`boolean markSupported()`	Sprawdza, czy strumień może obsługiwać metody `mark` i `reset`.
`abstract int read()`	Odczytuje kolejny bajt ze strumienia.
`int read(byte[] b)`	Odczytuje ze strumienia liczbę bajtów nie większą niż rozmiar tablicy `b`. Zwraca rzeczywiście odczytaną liczbę bajtów.
`int read(byte[] b, int off, int len)`	Odczytuje ze strumienia liczbę bajtów nie większą niż określona przez `len` i zapisuje je w tablicy `b`, począwszy od komórki wskazywanej przez `off`. Zwraca rzeczywiście przeczytaną liczbę bajtów.
`void reset()`	Wraca do pozycji strumienia wskazywanej przez wywołanie metody `mark`.
`long skip(long n)`	Pomija w strumieniu liczbę bajtów wskazywanych przez `n`. Zwraca rzeczywiście pominiętą liczbę bajtów.

Widać wyraźnie, że odczyt bajtów ze strumienia można przeprowadzić za pomocą jednej z metod o nazwie `read`. Przyjrzyjmy się metodzie odczytującej tylko jeden bajt. Mimo że odczytuje ona bajt, zwraca wartość typu `int`. Jest tak dlatego, że zwracana wartość zawsze zawiera się w przedziale 0 – 255 (to tyle, ile może przyjmować jeden bajt). Tymczasem zmienna typu `byte` reprezentuje wartości „ze znakiem" w zakresie od –128 do +127[1]. Spróbujmy zatem napisać prosty program, który odczyta wprowadzony z klawiatury znak, a następnie wyświetli ten znak oraz jego kod ASCII na ekranie. Kod realizujący przedstawione zadanie jest widoczny na listingu 6.1.

Listing 6.1.

```
import java.io.*;

public class Main {
  public static void main(String args[]) {
    System.out.print("Wprowadź dowolny znak z klawiatury: ");
    try{
      char c = (char) System.in.read();
      System.out.print("Wprowadzony znak to: ");
      System.out.println(c);
      System.out.print("Jego kod to: ");
      System.out.println((int) c);
    }
    catch(IOException e){
      System.out.println("Błąd podczas odczytu strumienia.");
    }
  }
}
```

[1] Oczywiście reprezentowanie zakresu od 0 do 255 byłoby możliwe również za pomocą typów short i long.

Pierwszym krokiem jest zaimportowanie pakietu `java.io` (lekcja 17.), znajduje się w nim bowiem definicja wyjątku `IOException`, który musimy przechwycić. W metodzie `main` za pomocą metody `System.out.println` wyświetlamy tekst z prośbą o wprowadzenie dowolnego znaku z klawiatury, a następnie wykonujemy instrukcję:

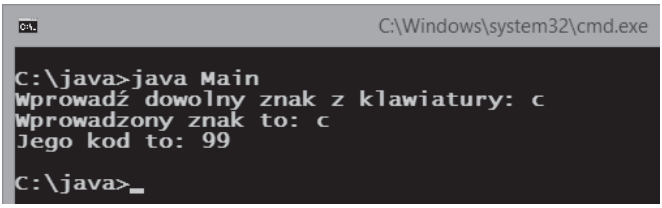
```
char c = (char) System.in.read();
```

Wywołujemy zatem metodę `read` obiektu `in` zawartego w klasie `System`[2]. Obiekt ten jest klasy `InputStream` (typem obiektu jest `InputStream`) i reprezentuje standardowy strumień wejściowy. W typowym przypadku jest on powiązany z klawiaturą, więc z tego strumienia będą odczytywane dane wprowadzane przez użytkownika z klawiatury. Metoda `read` zwraca wartość typu `int`, dokonujemy zatem konwersji na typ `char`, tak aby wyświetlić na ekranie wprowadzony znak, i przypisujemy go zmiennej `c`. Następnie wyświetlamy zawartość tej zmiennej za pomocą instrukcji:

```
System.out.println(c);
```

Aby wyświetlić kod znaku, należy ponownie dokonać konwersji. Zmienną `c` typu `char` konwertujemy na typ `int`, tak aby metoda `println` potraktowała ją jako liczbę, a nie jako znak. Ponieważ metoda `read` może spowodować powstanie wyjątku `IOException`, wszystkie instrukcje zostały ujęte w blok `try`, który zapobiegnie niekontrolowanemu zakończeniu programu (por. lekcje z rozdziału 4.). Przykładowy efekt działania programu z listingu 6.1 jest widoczny na rysunku 6.1.

Rysunek 6.1. *Przykładowy efekt działania programu z listingu 6.1*



Trzeba zwrócić uwagę, że w ten sposób tak naprawdę odczytywany jest ze strumienia nie jeden znak, ale jeden bajt. A te pojęcia nie są tożsame. Jeden znak może zawierać w zależności od standardu kodowania od jednego do kilku bajtów.

Wczytywanie tekstu

Wiadomo już, jak odczytać jeden bajt, co jednak zrobić, aby wprowadzić całą linię tekstu? Przecież taka sytuacja jest o wiele częstsza. Można oczywiście odczytywać pojedyncze znaki w pętli tak długo, aż zostanie osiągnięty znak końca linii, oraz połączyć je w obiekt typu `String`. Najlepiej byłoby wręcz wyprowadzić z klasy `InputStream` klasę pochodną, która zawierałaby metodę np. o nazwie `readLine`, wykonującą taką czynność. Trzeba byłoby przy tym pamiętać o odpowiedniej obsłudze standardów kodowania znaków. Na szczęście w JDK został zdefiniowany cały zestaw klas operujących na strumieniach wejściowych. Zamiast więc powtarzać coś, co zostało już zrobione, najlepiej po prostu z nich skorzystać.

[2] Obiekt `in` jest polem finalnym i statycznym klasy `System`.

Zadanie to będzie wymagało użycia dwóch klas pośredniczących. Klasą udostępniającą metodę `readLine`, która jest w stanie prawidłowo zinterpretować znaki przychodzące ze strumienia, jest `BufferedReader`. Aby jednak można było utworzyć obiekt takiej klasy, musi powstać obiekt klasy `Reader` lub klasy od niej pochodnej — w tym przypadku najodpowiedniejszy będzie `InputStreamReader`. Ponieważ nie ma takiego obiektu w aplikacji (do dyspozycji mamy jedynie obiekt `System.in` typu `InputStream`), należy go utworzyć, wywołując odpowiedni konstruktor. Będzie to obiekt (strumień) pośredniczący w wymianie danych. Do jego utworzenia potrzeba z kolei obiektu klasy `InputStream`, a tym przecież dysponujemy. Dlatego też kod programu, który odczytuje linię tekstu ze standardowego wejścia, będzie wyglądać tak, jak zostało to przedstawione na listingu 6.2.

Listing 6.2.

```
import java.io.*;

public class Main {
  public static void main(String args[]) {
    BufferedReader brIn = new BufferedReader(
      new InputStreamReader(System.in)
    );
    System.out.println("Wprowadź linię tekstu zakończoną znakiem Enter:");
    try{
      String line = brIn.readLine();
      System.out.print("Wprowadzona linia to: " + line);
    }
    catch(IOException e){
      System.out.println("Błąd podczas odczytu strumienia.");
    }
  }
}
```

Pierwszym zadaniem jest utworzenie obiektu `brIn` klasy `BufferedReader`. Mamy tu do czynienia ze złożoną instrukcją, która najpierw tworzy obiekt typu `InputStreamReader`, wykorzystując obiekt `System.in`, i dopiero ten obiekt przekazuje konstruktorowi klasy `BufferedReader`. Zatem konstrukcję:

```
BufferedReader brIn = new BufferedReader(
  new InputStreamReader(System.in)
);
```

można również rozbić na dwie części:

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader brIn = new BufferedReader(isr);
```

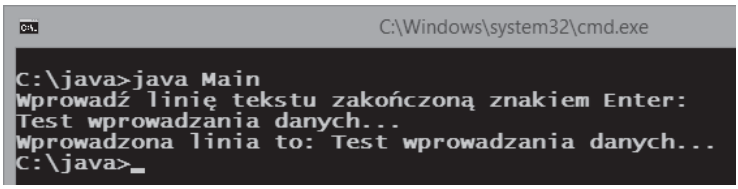
Znaczenie będzie takie samo, jednak w drugim przypadku zostanie utworzona zupełnie niepotrzebnie dodatkowa zmienna `isr` typu `InputStreamReader`. To, który ze sposobów jest czytelniejszy i zostanie zastosowany, zależy jednak od indywidualnych preferencji programisty. Z reguły stosuje się sposób pierwszy.

Kiedy obiekt klasy `BufferedReader` jest już utworzony, można wykorzystać metodę `readLine`, która zwróci w postaci obiektu typu `String` (klasa `String` reprezentuje ciągi

znaków) linię tekstu wprowadzoną przez użytkownika. Linia tekstu jest rozumiana jako ciąg znaków wprowadzony aż do naciśnięcia klawisza *Enter* (dokładniej: aż do osiągnięcia znaku końca wiersza w strumieniu). Wynika z tego, że instrukcja:

```
String line = brIn.readLine();
```

utworzy zmienną `line` typu `String` i przypisze jej obiekt (referencję do obiektu) zwrócony przez metodę `readLine` obiektu `brIn`. Działanie programu będzie zatem następujące: po uruchomieniu zostanie wyświetlona prośba o wprowadzenie linii tekstu, następnie linia ta zostanie odczytana i przypisana zmiennej `line`, a potem zawartość tej zmiennej zostanie wyświetlona na ekranie. Przykład wykonania tego programu pokazano na rysunku 6.2.



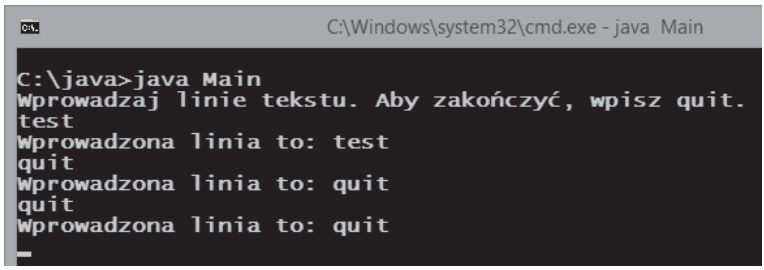
Rysunek 6.2. *Wczytanie linii tekstu za pomocą obiektu klasy BufferedReader*

Skoro wiadomo już, jak wczytać wiersz tekstu z klawiatury, spróbujmy napisać program, którego zadaniem będzie wczytywanie kolejnych linii ze standardowego wejścia tak długo, aż zostanie odczytany ciąg znaków `quit`. Zadanie wydaje się banalne, wystarczy przecież do kodu z listingu 6.2 dodać pętlę `while`, która będzie sprawdzała, czy zmienna `line` zawiera napis `quit`. W pierwszym odruchu napiszemy zapewne pętlę, która będzie wyglądać następująco (przy założeniu, że wcześniej został prawidłowo zainicjowany obiekt `brIn`):

```
String line = "";
while(line != "quit"){
  line = brIn.readLine();
  System.out.println("Wprowadzona linia to: " + line);
}
System.out.println("Koniec wczytywania danych.");
```

Warto przeanalizować ten przykład. Na początku deklarujemy zmienną `line` i przypisujemy jej pusty ciąg znaków. W pętli `while` sprawdzamy warunek, czy zmienna `line` jest różna od ciągu znaków `quit` — jeśli tak, to pętla jest kontynuowana, a jeśli nie, to kończy działanie. Zatem znaczenie jest następujące: dopóki `line` jest różne od `quit`, wykonuj instrukcje z wnętrza pętli. Wewnątrz pętli przypisujemy natomiast zmiennej `line` wiersz tekstu odczytany ze standardowego strumienia wejściowego oraz wyświetlamy go na ekranie.

Wszystko to wygląda bardzo poprawnie, ma tylko jedną wadę — nie zadziała zgodnie z założeniami. Aby się o tym przekonać, wystarczy uruchomić program z listingu 6.3, następnie wpisać kilka linii tekstu, a po nich ciąg znaków `quit`. Teoretycznie program powinien zakończyć działanie, tymczasem działa nadal, co jest widoczne na rysunku 6.3.



Rysunek 6.3. *Błąd w pętli while uniemożliwia zakończenie programu*

Listing 6.3.

```
import java.io.*;

public class Main {
  public static void main(String args[]) {
    BufferedReader brIn = new BufferedReader(
      new InputStreamReader(System.in)
    );
    System.out.println("Wprowadzaj linie tekstu. Aby zakończyć, wpisz quit.");
    String line = "";
    try{
      while(line != "quit"){
        line = brIn.readLine();
        System.out.println("Wprowadzona linia to: " + line);
      }
      System.out.println("Koniec wczytywania danych.");
    }
    catch(IOException e){
      System.out.println("Błąd podczas odczytu strumienia.");
    }
  }
}
```

Co się dzieje, gdzie jest błąd? Podejrzenie powinno najpierw paść na warunek zakończenia pętli `while`. On rzeczywiście jest nieprawidłowy, choć wydaje się, że instrukcja ta jest poprawna. Zwróćmy jednak uwagę na to, co tak naprawdę jest w tym warunku porównywane. Miał być porównany ciąg znaków zapisany w zmiennej `line` z ciągiem znaków `quit`. Tymczasem po umieszczeniu w kodzie sekwencji `"quit"` powstał nienazwany obiekt klasy `String` faktycznie zawierający ciąg znaków `quit`, a w miejsce napisu `"quit"` została wstawiona referencja do tego obiektu. Jak wiadomo z wcześniejszych lekcji (lekcja 13.), zmienna `line` jest także referencją do obiektu klasy `String`. Instrukcja porównania `line != "quit"` porównuje zatem ze sobą dwie REFERENCJE, które w tym przypadku muszą być różne. W związku z tym nie dochodzi tu do porównania zawartości obiektów. Dlatego program nie może działać poprawnie i wpada w nieskończoną pętlę (aby zakończyć jego działanie, trzeba wcisnąć na klawiaturze kombinację klawiszy *Ctrl+C*).

Koniecznie należy więc zapamiętać, że do porównywania obiektów nie używa się operatorów porównywania! Zamiast tego należy skorzystać z metody `equals`. Jest ona zadeklarowana w klasie `Object` i w związku z tym dziedziczona przez wszystkie klasy.

Warto pamiętać o tym podczas tworzenia własnych klas, by w razie potrzeby dopisać własną wersję tej metody.

Jak zastosować tę metodę w praktyce? Otóż zwraca ona wartość `true`, kiedy obiekty są takie same (czyli ich zawartość jest taka sama), lub `false`, kiedy obiekty są różne. W przypadku obiektów typu `String` wartość `true` zostanie zwrócona, jeśli ciąg znaków zapisany w jednym z nich jest taki sam jak ciąg znaków zapisany w drugim. Jeżeli np. istnieje zmienna zadeklarowana jako:

```
String napis1 = "test";
```

to wynikiem operacji:

```
napis1.equals("test");
```

będzie wartość `true`, a wynikiem operacji:

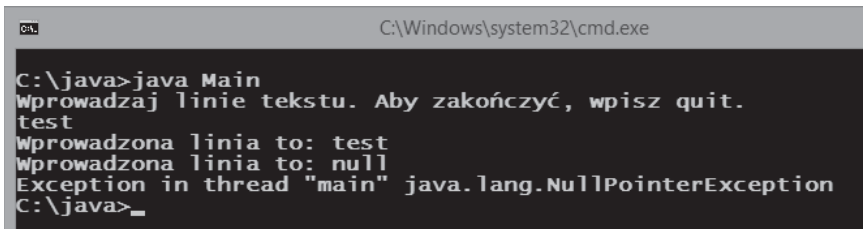
```
napis1.equals("java");
```

będzie wartość `false`.

Powróćmy teraz do pętli `while`. Jako że znane są nam już właściwości metody `equals`, w pierwszej chwili na pewno na myśl przyjdzie nam warunek:

```
while(!line.equals("quit")){
  /*instrukcje pętli while*/
}
```

Jeśli wprowadzimy go do kodu z listingu 6.3, aplikacja zacznie poprawnie reagować na polecenie `quit` — będzie się wydawać, że całość nareszcie działa prawidłowo. W zasadzie można byłoby się zgodzić z tym stwierdzeniem, gdyż program rzeczywiście działa zgodnie z założeniami, tylko że teraz zawiera kolejny błąd, tym razem dużo trudniejszy do wykrycia. Ujawni się on dopiero przy próbie przerwania pracy aplikacji. Spróbujmy uruchomić program, a po wpisaniu testowej linii tekstu przerwijmy jego działanie (w większości systemów należy wcisnąć kombinację klawiszy *Ctrl+C*). Efekt jest widoczny na rysunku 6.4. Zapewne nie spodziewaliśmy się wyjątku `NullPointerException`…



Rysunek 6.4. *Ukryty błąd w programie spowodował wygenerowanie wyjątku*

Na przyczynę powstania tego błędu naprowadza linia tekstu poprzedzająca komunikat maszyny wirtualnej, mianowicie: `Wprowadzona linia to: null`. Oznacza to bowiem (spójrzmy na kod), że metoda `readLine` obiektu `brIn` zwróciła — zamiast ciągu znaków — wartość `null`. Jest to standardowe działanie, metoda ta zwraca wartość `null`, kiedy zostanie osiągnięty koniec strumienia, czyli kiedy nie ma w nim już żadnych danych do odczytania. Co się dzieje dalej? Otóż wartość `null` jest przypisywana zmiennej `line`,

a potem w warunku pętli `while` następuje próba wywołania metody `equals` obiektu wskazywanego przez `line`. Tymczasem `line` ma wartość `null` i nie wskazuje na żaden obiekt. W takiej sytuacji musi zostać wygenerowany wyjątek `NullPointerException`.

Jak temu zapobiec? Możliwości są dwie: albo dodamy w pętli warunek sprawdzający, czy `line` jest równe `null`, a jeśli tak, to przerwiemy pętlę np. instrukcją `break`, albo też zmodyfikujemy sam warunek pętli. Ta druga możliwość jest lepsza, gdyż nie powoduje wykonywania dodatkowego kodu, a jednocześnie otrzymujemy ciekawą konstrukcję. Poprawiony warunek powinien wyglądać następująco:

```
while(!"quit".equals(line)){
  /*instrukcje pętli while*/
}
```

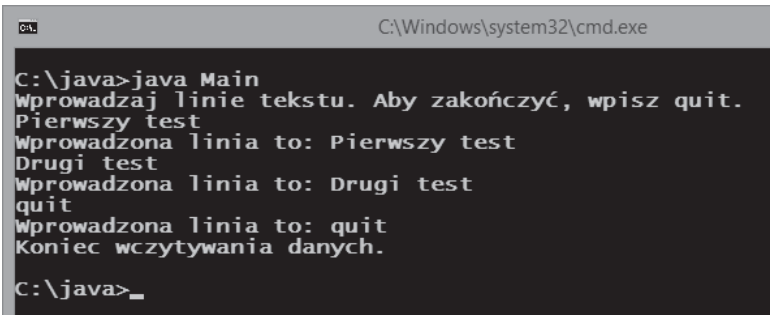
Początkowo tego typu zapis budzi zdziwienie: jak można wywoływać jakąkolwiek metodę na rzecz ciągu znaków? Wystarczy jednak sobie przypomnieć stwierdzenie, które pojawiło się kilka akapitów wyżej, otóż literał[3] `"quit"` w rzeczywistości powoduje powstanie obiektu klasy `String` i podstawianie w jego (literału) miejsce referencji do tego obiektu. Skoro tak, można wywołać metodę `equals`. To właśnie dzięki takiej konstrukcji unikamy wyjątku `NullPointerException`, gdyż nawet jeśli `line` będzie miało wartość `null`, to metoda `equals` po prostu zwróci wartość `false`, bo `null` na pewno jest różne od ciągu znaków `quit`. Ostatecznie pełny prawidłowy kod będzie miał postać przedstawioną na listingu 6.4. Przykład działania tego kodu został z kolei zaprezentowany na rysunku 6.5.

Listing 6.4.

```
import java.io.*;

public class Main {
  public static void main(String args[]) {
    BufferedReader brIn = new BufferedReader(
      new InputStreamReader(System.in)
    );
    System.out.println("Wprowadzaj linie tekstu. Aby zakończyć, wpisz quit.");
    String line = "";
    try{
      while(!"quit".equals(line)){
        line = brIn.readLine();
        System.out.println("Wprowadzona linia to: " + line);
      }
      System.out.println("Koniec wczytywania danych.");
    }
    catch(IOException e){
      System.out.println("Błąd podczas odczytu strumienia.");
    }
  }
}
```

[3] Czyli inaczej stała znakowa (napisowa), stały ciąg znaków umieszczony w kodzie programu.

Rysunek 6.5. *Wprowadzanie w pętli kolejnych linii tekstu*



Wprowadzanie liczb

Potrafimy już odczytywać w aplikacji wiersze tekstu wprowadzane z klawiatury, równie ważną umiejętnością jest wprowadzanie liczb. Jak to zrobić? Trzeba sobie uzmysłowić, że z klawiatury zawsze wprowadzany jest tekst. Jeśli próbujemy wprowadzić do aplikacji wartość `123`, to w rzeczywistości wprowadzimy trzy znaki, `"1"`, `"2"` i `"3"` o kodach ASCII 61, 62, 63. Mogą one zostać przedstawione w postaci ciągu `"123"`, ale to dopiero aplikacja musi przetworzyć ten ciąg na wartość `123`. Takiej konwersji w przypadku wartości całkowitej można dokonać np. za pomocą metody `parseInt` z klasy `Integer`. Jest to metoda statyczna, można ją więc wywołać, nie tworząc obiektu klasy `Integer`[4]. Przykładowe wywołanie może wyglądać następująco:

```
int liczba = Integer.parseInt(ciąg_znaków);
```

Zmiennej `liczba` zostanie przypisana wartość typu `int` zawarta w ciągu znaków `ciąg_znaków`. Gdyby ciąg przekazany jako argument metody `parseInt` nie zawierał poprawnej wartości całkowitej, zostałby wygenerowany wyjątek `NumberFormatException`. Aby zatem wprowadzić do aplikacji wartość całkowitą, można odczytać wiersz tekstu, korzystając z metody `readLine`, a następnie wywołać metodę `parseInt`. Ten właśnie sposób został wykorzystany w programie z listingu 6.5. Zadaniem tego programu jest wczytanie liczby całkowitej oraz wyświetlenie wyniku mnożenia tej liczby przez wartość `2`.

Listing 6.5.

```
import java.io.*;

public class Main {
  public static void main(String args[]) {
    BufferedReader brIn = new BufferedReader(
      new InputStreamReader(System.in)
    );

    System.out.print("Wprowadź liczbę całkowitą: ");

    String line = null;
    try{
      line = brIn.readLine();
    }
```

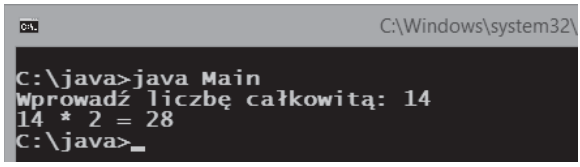
[4] Jak widać, metody statyczne, które były przedmiotem lekcji 18., przydają się w praktyce.

```
        catch(IOException e){
          System.out.println("Błąd podczas odczytu strumienia.");
          return;
        }

        int liczba;

        try{
          liczba = Integer.parseInt(line);
        }
        catch(NumberFormatException e){
          System.out.print("Podana wartość nie jest liczbą całkowitą.");
          return;
        }
        System.out.print(liczba + " * 2 = " + liczba * 2);
      }
    }
```

Kod zaczyna się od utworzenia obiektu `brIn` klasy `BufferedReader` powiązanego poprzez obiekt pośredniczący klasy `InputStreamReader` ze standardowym strumieniem wejściowym. Stosujemy technikę opisaną przy omawianiu przykładu z listingu 6.2. Następnie wyświetlamy prośbę o wprowadzenie liczby całkowitej oraz odczytujemy wiersz tekstu za pomocą metody `readLine` obiektu `brIn`, nie zapominając o przechwyceniu ewentualnego wyjątku `IOException`. W przypadku wystąpienia takiego wyjątku wyświetlamy stosowną informację na ekranie oraz kończymy wykonywanie programu. Warto zwrócić uwagę, że używana jest w tym celu instrukcja `return` bez żadnych parametrów, co oznacza wyjście z funkcji `main` (bez zwracania wartości), a tym samym zakończenie działania aplikacji.

Jeśli odczyt wiersza tekstu się powiedzie, zostanie on zapisany w zmiennej `line`. Deklarujemy więc dalej zmienną `liczba` typu `int` oraz przypisujemy jej wynik działania metody `parseInt` z klasy `Integer`. Metodzie przekazujemy ciąg znaków zapisany w zmiennej `line`. Jeśli wprowadzony przez użytkownika ciąg znaków nie reprezentuje poprawnej wartości liczbowej, wygenerowany zostaje wyjątek `NumberFormatException`. W takim wypadku wyświetlamy komunikat o tym fakcie oraz kończymy działanie funkcji `main`, a tym samym programu, wywołując instrukcję `return`. Jeśli jednak konwersja tekstu na liczbę powiedzie się, odpowiednia wartość zostanie zapisana w zmiennej `liczba`, można zatem wykonać mnożenie `liczba * 2` i wyświetlić wartość wynikającą z tego mnożenia na ekranie. Przykładowy wynik działania programu jest widoczny na rysunku 6.6.

Rysunek 6.6. *Przykładowy wynik działania programu z listingu 6.5*



Gdyby miała być wczytana liczba zmiennoprzecinkowa, należałoby do konwersji zastosować metodę `parseDouble` z klasy `Double` lub `parseFloat` z klasy `Float`, co jest doskonałym ćwiczeniem do samodzielnego wykonania.

Skorowidz

E

F

G

H

I

J

K

L

M

N

O

P

R

S

Ś

T

U

W

Z

PRAKTYCZNY KURS

Znajomość Javy, jako jednego z najpopularniejszych języków programowania na świecie, to dziś obowiązek każdego programisty. Na nic nie zdadzą się wykręty, że przecież można inaczej — pamiętaj, Java jest wszędzie! Dlatego jeśli masz ambicje zająć się programowaniem albo szybko nauczyć się podstaw nowego języka, natychmiast musisz zaopatrzyć się w tę książkę. Pomoże Ci ona zrozumieć składnię i logikę Javy, a także opanować polecenia, byś mógł szybko rozpocząć samodzielną pracę w tym języku.

Z książki dowiesz się, jak zainstalować Javę i jakie narzędzia będą Ci potrzebne. Odkryjesz podstawowe zasady pisania programów z uwzględnieniem obiektowego charakteru tego języka, nauczysz się stosować instrukcje i odpowiednie typy danych, a także wykorzystywać deklaracje, klasy i metody. Kolejne rozdziały wskażą Ci, do czego służą pętle, tablice i operatory. Zanim się obejrzysz, będziesz już umiał dokonywać konwersji typów, używać konstruktorów i zagnieżdżać klasy, a stąd już tylko krok do tworzenia apletów i aplikacji. Przekonaj się, jak szybko możesz poznać Javę — wystarczy seria konkretnych ćwiczeń, a Twoje umiejętności poszybują w górę!

- Krótka historia Javy, narzędzia i wersje
- Struktura programu, kompilacja i wykonanie
- Podstawy obiektowości, typy danych, komentarze, deklaracje i przypisania
- Wyprowadzanie danych na ekran i operacje na zmiennych
- Instrukcje if...else, break i continue, switch i operator warunkowy
- Pętle, podstawowe operacje na tablicach i tablice wielowymiarowe
- Klasy, pola, metody, argumenty i przeciążanie metod
- Konstruktory, klasy potomne, specyfikatory dostępu i pakiety
- Przesłanianie metod, składowe statyczne, klasy i składowe finalne, blok try...catch
- Konwersje typów, rzutowanie obiektów, klasy abstrakcyjne i tworzenie interfejsów
- Rodzaje klas wewnętrznych, dziedziczenie, klasy anonimowe i zagnieżdżone
- Standardowe wejście i wyjście, system plików, operacje na plikach
- Kontenery, typy uogólnione, podstawy apletów
- Kroje pisma (fonty) i kolory, grafika, dźwięki i obsługa myszy
- Tworzenie aplikacji i komponenty

Napisz swój pierwszy program w Javie!

sięgnij po WIĘCEJ

KOD KORZYŚCI

Helion

26437 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-246-9663-5

9 788324 696635

Informatyka w najlepszym wydaniu

cena: 69,00 zł